DOCUMENT RESUME

ED 052 482	24	CG 006 490

AUTHOR	Taylor, Ann
TITLE	GLURP-Generalized Language for Understanding and Responding to People. Information System for Vocational Decisions.
INSTITUTION	Harvard Univ., Cambridge, Mass. Graduate School of Education.
SPONS AGENCY	Office of Education (DHEW), Washington, D.C.
REPORT NO	PR-29
BUREAU NO	BR-6-1819
PUB DATE	Jan 70
GRANT	OEG-1-6-061819-2240
NOTE	137p.
EDRS PRICE	EDRS Price MF-$0.65 HC-$6.58
DESCRIPTORS	Career Choice, *Computer Assisted Instruction, Educational Technology, *Information Systems, *Man Machine Systems, *Occupational Guidance, *Programing Languages, Programing Problems

ABSTRACT

GLURP (Generalized Language for Understanding and Responding to People) is a mid-point version of a script-writing computer language which had been intended by the developers of the Information System for Vocational Decisions (ISVD). Because ISVD aimed at allowing the student user to control the order and quantity of data flow (in an English language context) to the console certain requirements were demanded of the author language. This paper describes: (1) the current status of ISVD's author language (GLURP); (2) its adaptation to machines other than the RCA Spectra or IBM 360 on which it can presently be used; and (3) some system and language problems which were not solved due to insufficient time. Other computer languages used in constructing GLURP are discussed in the appendices. (TL)

ED052482

INFORMATION SYSTEM FOR VOCATIONAL DECISIONS

Project Report No. 29

GLURP -- GENERALIZED LANGUAGE for UNDERSTANDING and RESPONDING to PEOPLE

Ann Taylor

This paper was supported in part by Grant No. OEG-1-6-061819-2240
of the United States Office of Education under terms of the
Vocational Education Act of 1963.

Graduate School of Education
Harvard University

January 1970

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY

CG 006 490

ED052482

GLURP -- GENERALIZED LANGUAGE for UNDERSTANDING and RESPONDING to PEOPLE

GLURP is a mid-point version of what we set out to accomplish as a computer language for social science/educational applications of the computer. The aim of the ISVD project has been to allow the student user to control the order and quantity of data flow to the console, but still in an English language context. The ISVD author language, then, must be able to accept student input, convert it into information to the script, which then sends to the student appropriate data from the data base for the student to convert into information. This is different from the usual CAI language, in that the student is never "right" or "wrong" -- indeed, his state should be described as "curious" or "fascinated". Also, other CAI languages have their data preset in the text, following a preset order as in normal classroom presentation (but without a question period). There is usually no provision for random access of data. Most teaching languages, then, are really lecturing or drill-and-practice languages.

In our investigations we discovered two languages, each of which was capable of one of our main objectives. ELIZA (see Appendix B) emphasized the understanding by the script of what the student was trying to say. TRAC* (Appendix A) emphasized the ability to convert English input into information usable by the script. Combining these two languages with a random access data routine (HASM/DASM) gave us most of the language capabilities required, but did not suggest a homogeneous format for the author language.

What we have accomplished in our allotted time, then, could be carried on in one of two ways: the system could be taken as is (or with minor interface modifications) to be used on an RCA Spectra or IBM 360 machine, or the whole thing could be re-written for another machine. In the second case, I would strongly suggest that the heirs of the system study what we have done and make an attempt to evolve the language and data access routines closer to the ideal before starting the coding.

The rest of this paper, accordingly, will be in three parts: 1) a description of GLURP as it exists, 2) a description of how to adapt it to another BAL-oriented machine, and 3) some problems we would have attempted to solve had we had more time.

---

*TRAC is the trademark of Rockford Research Institute, Inc., Cambridge, Massachusetts, for its string handling language.

I. A DESCRIPTION OF GLURP AS IT NOW EXISTS

The basic control mechanism for GLURP is the SCRIPT, which is, in well-written scripts, controlled by the student at the console. All data, including scripts, are accessed by specifying the name of the file, the name of the record within that file, and the name of the attribute within the record. All of these names look rather like English in the hope that key words used by the student may be used directly to seek information from the disc.

A script resides on the disc in one of the two script files (SCRIPTSA and SCRIPTSB) and is accessible by name through the HASM routine. The script names are record names to HASM and the step numbers are attribute names.

In the existing system, all script names have no embedded blanks because of the way the TRAC "pop-top" function works, and the fact that PT is used extensively in linking from one script to another. More about that in section III.

GLURP sets things in motion by calling for the first step of the LOGIN script. This script is responsible for identifying the student and for calling HASM to bring his personal files into play. See document on script network for further description of script control.

The script is read in, step by step, interpreted and executed. Some steps will cause the sequential interpretation steps to be altered and control restarted with a new script and/or step.

Each step consists of one of the two types of GLURP statements, a TRAC statement or a non-TRAC statement. A TRAC statement (which begins with a "#") may lead to a very complex set of operations, whereas a non-TRAC statement (not beginning with a "#") usually refers to a rather simple sort of operation. The exception to this is the non-TRAC "DECOMP" statement which is actually a blend of ELIZA and TRAC.

1. A BRIEF DESCRIPTION OF TRAC

A basic unit of TRAC is the "FUNCTION". A function describes something the machine should do. A function can be recognized as something that begins with the two characters

#(

and ends with the character

)

The function is broken down into parts called "ARGUMENTS", which are separated by colons. Here is a TRAC function

```
#(PS:HELLO THERE)
```

It has two arguments: PS and HELLO THERE.

The argument of a function may be or may contain yet another TRAC function, which in turn may contain a TRAC function, etc. A combination or series of functions or things to do is called a "PROCEDURE". A script is a GLURP procedure, consisting of several steps of GLURP statements. This is a TRAC procedure

```
#(PS:#(MESSAGE))#(GO:010.00)
```

The words "STRING" and "LIST" are used often in TRAC and GLURP. A string is any sequence of characters, be it a sentence, a word, a paragraph, a function, or a GLURP statement. A list is a series of words put together, separated by blanks (a "WORD LIST") such as

doctor lawyer indian chief

or by commas (a "RULE LIST"), such as

0 YES 0, 0 OKAY 0, 0 SUPPOSE SO 0

Let us leave it for now that such things exist; the use of them will become clear as we progress.

When dealing with a computer, we keep data in two types of storage areas. Most of our data is on the disc. It takes a certain amount of effort for the machine to get material off the disc, so as much as possible it likes to keep things closer at hand in "core storage." Before the data can be used it must be brought into core storage. We'll talk about that later. But no matter where the data is stored, the script writer must put it there explicitly before he tries to use it, or know that someone else has put it there for him. You cannot use a string that is not there. Let's go back to our first example.

```
#(PS:HELLO THERE)
```

If we look up the PS function in Appendix C, we find that it is the "print string" function. It has two arguments, and will print the second one on the teletype (or CRT, whichever you have). All the data we need, in this case, is right in the function. Suppose, however, that you had a big long paragraph of material to print, and that you wanted to refer to it in several different places. Instead of writing it all out each time in a PS function, it would be nice to tell the machine what the big long message is, and then tell the PS function to use that previously defined string. We do it this way:

```
#(DS:MESSAGE:BIG LONG MESSAGE)
```

This stores away in core storage a string by the name of MESSAGE whose contents is the BIG LONG MESSAGE. (see Appendix C, DS function) Now we can refer to this string at any time with the notation

```
#(MESSAGE)
```

This should be read "the contents of string MESSAGE." As mentioned above, the argument of a function may in turn be a function, so we may write

```
#(PS:#(MESSAGE))
```

which may be read "print the contents of the string MESSAGE." The result of using this function would be to have the text

```
BIG LONG MESSAGE
```

typed on the teletype (or CRT).

Note that by stating a string name rather than a TRAC function name as the first argument of a function, we can call a string out of core storage. This means, of course, that we must not give a string the same name as a function name. We defined a function as a statement of something the machine should do. A function may do any combination of these types of things:

1. Put something or get something from external sources (disc or console).
2. Change or manipulate strings in core storage, and/or
3. Perform a calculation which results in an answer.

If a function performs a calculation which results in an answer, it is called a "VALUED" function; otherwise, it is called an "UNVALUED" function. When a function has a value, it replaces itself with that value in the evaluation of a procedure.

For example, the function

```
#(MESSAGE)
```

in the procedure

```
#(PS:#(MESSAGE))
```

has the value

```
BIG LONG MESSAGE
```

When the procedure is being used, then, this function replaces itself in the procedure and leaves

```
#(PS:BIG LONG MESSAGE)
```

The function is re-evaluated in respect to the surrounding or "deferred" function, and the PS function is carried out. None of the functions we used, however, changes core storage in any way, so the string MESSAGE remains unchanged in core. If a function is unvalued, it does its designated task and then disappears from the procedure being evaluated. To understand this better, we need to understand something of how the TRAC interpreter behaves.

In effect, what happens is that TRAC scans a procedure from left to right until it finds a right paren. At this point it scans backwards to the last #(configuration and analyzes the function found within. The value of the analyzed function, if any, replaces the function in the procedure to analyze any further functions. If argument 1, the function name or string name, does not exist, nothing happens and the function is erased. If the entire procedure is scanned and no TRAC functions remain, a new procedure is brought in from the next script step and the process begins all over again. For example, let's take the procedure

```
#(DS:COUNT:1)#(PS:#(AD:#(COUNT):1))
```

The interpreter scans along until it finds the first right paren. It then proceeds to evaluate the function

```
#(DS:COUNT:1)
```

When it is finished with this function, a string has been set up in core storage with the name COUNT and contents 1. According to the description in Appendix C, DS is an unvalued function, so it now erases itself from the procedure leaving

```
#(PS:#(AD:#(COUNT):1)
```

The first right paren this time leaves us just after the word COUNT, so the interpreter picks out the function

```
#(COUNT)
```

This function replaces itself with the contents of string COUNT in the procedure, leaving

```
#(PS:#(AD:1:1))
```

Further scan leads us to the function

```
#(AD:1:1)
```

This function replaces itself with the value 2. Note that none of the functions used change core storage, so we still have stored a string called COUNT with contents 1. Now we have left the procedure

```
#(PS:2)
```

and "2" is printed on the teletype (CRT).

Sometimes it is handy to store some strings that are TRAC functions or procedures in themselves. But if we were to try to do so in this fashion

```
#(DS:ROUTINE:#(PS:HELLO THERE))
```

we know from the above example that this would print HELLO THERE on the teletype and define a string ROUTINE with no contents at all. For this reason TRAC uses parens as protectors for things it doesn't want to have analyzed. We would define the string this way

```
#(DS:ROUTINE:(#(PS:HELLO THERE)))
```

The scanning routine finds the left paren not preceded by a #. It now skips to its matching right paren, saving but not analyzing the material in between. In this process, it strips out the protecting parens. When we call back our string with the function

```
#(ROUTINE)
```

the function is replaced by its value

```
#(PS:HELLO THERE)
```

which is further scanned to produce the message HELLO THERE on the teletype. But now suppose that we wanted to print out the contents of the string ROUTINE. If we tried using the procedure

```
#(PS:#(ROUTINE))
```

we would receive the message

```
HELLO THERE
```

Again, TRAC has provided for this and has a way of saying "evaluate this function, but if the value of the function is in turn a function, do not evaluate it further." We do this by typing

```
#(PS:##(ROUTINE))
```

The string ROUTINE is evaluated once to produce

```
#(PS:#(PS:HELLO THERE))
```

but, since we used the function sign ##, does not evaluate it further and goes directly to its surrounding function PS. The message on the teletype is now

```
#(PS:HELLO THERE)
```

which is the contents of the string ROUTINE.

In Appendix C, the various arguments of a function are described by various letters, depending on what type of thing the argument must be. The first argument of a function is always a two-letter function code, telling what the function is, and should be written just as it appears in the description. The rest of the arguments should be interpreted as follows:

X means the argument must be a string. This, of course, can be represented by a valued TRAC function or procedure. This may also be no characters at all if this is logical at the time.

N represents the name of a string. Again, this can be represeted completely or partly by a valued TRAC function.

D represents a number, with no alphabetics or decimal point included. A negative number is preceded by a minus sign.

B represents a Boolean string. See BA function in Appendix C.

Z represents either a string, as in X, or a TRAC procedure or routine surrounded by parens. In the ISVD system it can also be a step number of the current script, preceded by an asterisk, as in

```
#(EQ:#(NAME):HENRY:*100.00:*200.00)
```

According to Appendix C, this says "if the string NAME contains the word HENRY, go to statement 100.00, otherwise go to statement 200.00." This could also be written

```
#(EQ:#(NAME):HENRY:(#(GO:100.00)):(#(GO:200.00)))
```

which reads exactly the same. Note the parens around Z arguments that are TRAC functions, and remember what would happen if those parens were not there. It is possible, however, that the Z could be a simple string

```
#(AD:#(EQ:#(COUNT):3:1:2):1)
```

This says, if the contents of COUNT is 3 add 1 and 1, otherwise add 2 and 1.

2. A STUDENT COMMAND LANGUAGE

Pincus, Yee and Little proposed several additions to the script software. These serve the functions of (1) giving the students control over the direction of the script flow, and (2) making these an automatic part of the KEYBOARD statement, so that each script does not have to check for the occurrence of these student commands.

As the student enters the system, he is given an INTRODUCTION script (as contrasted to the ORIENTATION script) that explains the mechanical use of the machine, and also certain commands he can use at any time to change the subject, ask questions of the data base, etc. The student commands are:

@STOP This causes the script to stop whatever it is doing and to return to the latest named STOP linkage point. Script writers should anticipate the use of this command by inserting at appropriate points the place a student should return to if he should suddenly break out of a script in this manner. Usually, a stop return point will be a point where he is given a choice of several different activities. He may then make a choice, try that activity for a while, then say @STOP because he would rather choose something else to do.

@QUIT This causes immediate transfer to the QUIT script, which asks him if he wants a summary of his activities to date. He is given a summary or not, as he chooses, and the machine prepares for a new user.

@DATA Sends control directly to a script giving him direct access to any data base. At this point he may ask for specific pieces of data at will until he types @STOP, when he is returned to the CRT statement immediately preceding the KEYBOARD statement at which he typed @DATA. Note that this is a non-typical use of @STOP, in that the author need not worry about this occurrence . . . the GLURP interpreter will take care of it automatically.

@HELP If the student doesn't understand a question or for some other reason doesn't know what to respond, he types @HELP, whereupon he is sent to a section of the script which further explains what the author was trying to say. It is the responsibility of the script writer to specify, on the KEYBOARD statement, where the student is to be sent if he types @HELP. If no help has been specified, the student is told "No hint has been provided. Answer the best that you can."

@SUMMARY Sends the student to the SUMMARY script, where he is given a summary of his activities to date. As in @DATA, when he types @STOP, he is returned to the CRT statement immediately preceding the KEYBOARD statement at which he typed @SUMMARY.

3. WRITING A SCRIPT IN GLURP

In GLURP step numbers have the format XXX.XX, where the X's are digits. The maximum step is 999.99. I suggest that scripts be initially written with gaps between the step numbers to allow for future expansion of the scripts, since these numbers must be in order of increasing value.

A script, then, consists of a series of GLURP statements, each beginning with a statement number, and each consisting of either a GLURP statement or a TRAC statement. *Note, however, that these two statement types cannot be mixed in one statement. In any case, a statement cannot be longer than 255 characters.*

There are currently four GLURP functions that are non-TRAC:

1. CRT or CRT+ or CRTC
2. KEYBOARD
3. DECOMP
4. GOTO

I hear cries about the hard copy or printer device. Unfortunately, this will be used in the following way: If the student wants a printout of the current contents of the CRT, he pushes the "hard copy" button and waits till the printing is finished before he continues with the interaction. Formatting characters for the CRT (see CRT description) will have no effect on the teletype, . . . an important consideration if you are planning the transfer of a table, formatted list, blinking characters, etc.

This is not the only button the student will be faced with: in order to have a slide shown, not only must there be a SLIDE command in a CRT statement, but the student must be instructed to push the "run" button in order to get it to show. There was some indication trom the Sanders people that they could rig things so that the computer could push the "run" button, but that isn't the way it is now.

Here are descriptions of the GLURP statements in full.

CRT or CRT+ or CRTC

CRT causes the following text to be displayed with carriage returns inserted as close to the end of the maximum sized line as possible. The text will start in the upper left hand corner of the screen, and all previously displayed material will be erased. The text may not contain the special characters > or \ except in the case of a slide command, and it must be 250 characters or less.

CRT+ causes the following text to be added to whatever is currently on the screen, starting with the next available line. Again, the carriage returns are automatically inserted at the end of the 64 character line. The maximum combined length of the characters of the screen is 768 characters, *including any concurrent slide commands.*

CRTC causes material to be added to whatever is on the screen beginning at the next available space +1.

The Sanders 720 formatting characters are multipunched in one position:

X'6A' (home) causes the following text to start in the upper left hand corner of the screen. Previously displayed text is NOT erased.

X'B9' (clear, or initiate) erases the contents of the screen and acts as a $\overset{H}{M}$ character.

X'5F' (carriage return) causes the following text to appear on the next available line, starting with the left-most character.

X'4A' (horizontal tab) causes four blanks.

X'4F' (vertical tab) causes the following text to start four lines down from its current position and at the left-most position of that line.

X'BF' (start blink) causes the following characters, up to the clear blink format character, to blink as they are displayed. This is a good substitute for underlining.

X'70' (clear blink) causes the following text not to blink.

N.B. *There is no timer associated with the CRT. This means that if the CRT statements are presented in a series, the text will flash by at faster-than-readable speed. For this reason, each CRT or series of CRT and its associated CRT+'s must be followed by a KEYBOARD statement.* The script will then pause until the student has finished reading the message and pushed "send block."

The slide command is actually a specialized CRT+ command. The characters that specify the slide sequence are sent to the CRT where it is saved but not displayed. (It is printed in the case that the student requests hard copy, however!) The student must then be told to push the "run" button to get the slide projector to search the screen for its command. The slide commands cause the specified slide(s) to be shown for the specified length of time. The exact format for a slide designation is

```
\S0X/MSS/NNN/NNN///*
```

where X is the projector number (1 or 2), M is the number of minutes the slide should be shown, SS is the number of seconds the slide should be shown, NNN is a slide number. The asterisk marks the end of the command. If MSS is 000, the slide will be shown until another slide replaces it, or the slide projector is turned off. The projector is turned off with the instruction

```
\S0X//////*
```

If a series of slides are to be shown, say slides 5 through 10, for 5 seconds each, the instruction would be

```
\S0X/005/005/010///*
```

After this series is shown, the projector will look for another command without the student pushing the "run" button.

If one only slide is to be shown, say slide 7 for 1 minute and 3 seconds, the instruction would be

```
\S0X/103/007////*
```

If there is some text that you wish displayed only while this particular slide is being shown, the format is

```
\S0X/103/007////*[THIS IS JOHNNY BEING A DOCTOR]
```

The text must be included in the maximum number of characters that can be displayed at once, but it will actually be displayed only when slide 7 is being displayed.

KEYBOARD

Prints three asterisks in the student response area (block 1, last 256 positions of the CRT) and reads the student response. All responses are automatically checked for student commands (leading @) or TRAC (leading #). A branch point on the KEYBOARD statement

```
KEYBOARD:XXX.XX
```

indicates which statement the student should be transferred to if he types @HELP. If the branch point is omitted, and the student types @HELP, he will be informed that no hint has been provided.

DECOMP:rule list:stepno list

Analyzes the last student response. This is probably best explained by example. Let's say you were interested in finding out if, at the last KEYBOARD statement, the student answered "yes". You would write

```
DECOMP:YES:004.00
```

This says, if the last response was exactly "yes", transfer to statement 004.00, otherwise go to the next statement. The above example has a rule list with one item in it -- YES. Usually, though, you would be more interested in whether or not the student used the word "yes" with accompanying text. In this case you would write

```
DECOMP:0 YES 0:004.00
```

This says if the last response was any number of words (including no words at all) followed by "yes", followed by any number of words, transfer to statement 004.00. This is still too exact, however, and what you really want to know is whether or not the student answered in a positive fashion. In this case you would previously have had a statement

```
#(DS:POS:0 YES 0,0 OKAY 0,0 THINK SO 0, etc...)
```

This defines a string which is a "rule list" . . . a series of decomposition rules to be referred to in a DECOMP statement. Then, when it came time to analyze the response, you would write

```
DECOMP:(POS):004.00
```

This says, if any of the rules in the POS string apply to the last response, go to statement 004.00.

Now let's try to see if the responses were positive, negative, or unsure. You could write

```
DECOMP:(POS),(NEG),(UNSURE):004.00:006.00:030.00
```

This says if the ans er is positive, go to statement 004.00, if it is negative go to statement 006.00, and if it is unsure, go to statement 030.00. We can use the strings POS, NEG, and UNSURE because they are permanently defined.

Now suppose you want to use a rule list only once, and don't want to go to the bother of defining an external string. Say, for example, if the student types A, B, or C, you want to go to statement 006.00, but if he types D you want to go to statement 010.00. The statement is

```
DECOMP:(*A B C),D:006.00:010.00
```

The element (* A B C) is an "internal word list." A word list is different from a rule list, in that each item in the list contains only one word (no phrases or 0's), and that, in DECOMP statements, it is interpreted to mean "any one of these things will do." It may be the case that, during the execution of a script, you have collected from the student a word list of, say, his interests. You have called this list INTERESTS, and have collected it with the aid of TRAC functions such as NT, NB, etc. (see Appendix C) It looks like this: SKIING SWIMMING READING MATH. Now you want to see if, in the last response, the student mentioned one of those interests. You would write this

```
DECOMP: 0 (/INTERESTS) 0:004.00
```

This tells the machine that the list involved is a simple list, not a collection of rules. If the student input is any number of words followed by either SKIING or SWIMMING or READING or MATH, followed by any number of words, the rule fits. This time suppose that you want to check for a positive response, but if the student says "I think so," you want to consider that an unsure-type response. You can still use the same POS string, but write branch points directly in the call for an external rule list. An asterisk says, if everything matches so far, keep going with the rest of the rule; a statement number says if everything matches this far, go to this statement without trying to match the rest of the rule. If you have put neither asterisks or statement numbers as in (POS), it says if everything matches till here, keep going with the rest of the rule. In other words, it is as if you had written (POS:*:*:*: etc.) For example,

```
DECOMP:(POS:*:*:010.00:etc...):004.00
```

This says, apply the POS rule list to the response, and if the third rule applies go to statement 010.00 otherwise if a rule fits go to statement 004.00.

To be even more elaborate, let's say you want to analyze a response, but you want to make sure there isn't a "not" or "don't" or other negative word that would reverse the meaning of the response. You would define a rule string

```
#(DS:NOT:((NEG),0))
```

Note that the second rule of the list allows for any response whatsoever. Now you use it this way

```
DECOMP:0 (NOT:013.00:*) 0 PLAY 0 (NOT:013.00:*) 0:008.00
```

This says, if the response has negatives in it, go to statement 013.00, otherwise, if it has the word "play" in it, go to statement 008.00, and if none of these fit, go to the next statement. Note that rule lists may refer to other rule lists, which may refer to other rule lists, etc.

GOTO XXX.XX

Transfers control to the designated statement number.

4. GETTING AND STORING DATA FROM THE DISC

Every piece of data in the system is labeled by a three-part name:

```
FILE:RECORD:ATTRIBUTE
```

FILE refers to one of the 9 data bases. The user data file is a personal file, containing information about the current user, his progress through the system, information gathered from him by the system. The use of the TRAC LI function causes the current user's private file name to be stored in a string called NICK while he is using the console.

RECORD refers to a subsection of a file. In the occupation or military file, RECORD would be a job title; in Education it would be a school name, etc.

ATTRIBUTE is a subdivision of RECORD, such as SALARY or RELATED CAREERS under DOCTOR, or TUITION under HARVARD.

The TRAC format for retrieving data is

```
#(FR:FILE:RECORD:ATTRIBUTE:N:Z)
```

for example

```
#(FS:EDUCATION:HARVARD:TUITION:VAR1:*020.00)
```

This would retrieve the tuition for Harvard and store it as a string with the name VAR1. If the record cannot be found, control is transferred to statement 020.00.

Should you want to simply have the piece of data inserted into some text you would write

```
#(PS:text #(FR:EDUCATION:HARVARD:TUITION::020.00) text.)
```

leaving out the string name parameter, but not its colon. In this case, if the data cannot be found, control would go to statement 020.00 without any printing having occurred.

The TRAC format for storing data is similar.

```
#(SR:FILE:RECORD:ATTRIBUTE:SOME DATA:Z)
```

where SOME DATA is the actual thing you want to store under the specified three-part name.

At this point it would be useful to become familiar with two more TRAC functions, dealing with the student's last response. The first of these is

```
#(CK)
```

This is the "call keyboard" function, and it has the value of whatever the student typed into the machine last. If an answer is to be saved for use further on in a script, you would do it with a

```
#(DS:ANSWER:#(CK))
```

kind of statement.

The other function is

```
#(KS:D)
```

This is the "keyboard segment" function, and its value is the Dth segment of the student response, according to the last DECOMP rule applied to it. You will recall that D stands for a number. For example, the rule

```
0 BECAUSE I 0 SO 0
```

applied to the response

```
BECAUSE I REALLY DON'T THINK SO AT ALL
```

would result in the following segments

Segment 1	nothing
Segment 2	BECAUSE I
Segment 3	REALLY DON'T THINK
Segment 4	SO
Segment 5	nothing at all

Either of these functions may be used to specify the data in the SR segment

```
      #(SR:FILE:RECORD:ATTRIBUTE:#(CK):Z)
or    #(SR:FILE:RECORD:ATTRIBUTE:#(KS:D):Z)
```

Sometimes asking for a specific piece of data is not enough. There is also the idea of asking for the names of all those records in a given file that have a certain description:

```
#(FR:OCCUPATIONS:WORK OUTDOORS:LIST:TEMP:*20.00)
```

This would put into TEMP a list of those names of jobs that require working outdoors. Note that work outdoors is an actual record name, and script writers must be sure that such a record exists before they request it. If you expect a rather long list, say up to 853 job titles, you will probably want to get a coded or "BOOLEAN" version of the list.

```
#(FO:OCCUPATIONS:WORK OUTDOORS:LIST:TEMP)
```

In the data base, the record WORK OUTDOORS, for example, (or any record that is actually a list) is represented by 853 digits that are either 1 (if outdoors work is required) or 0 (if outdoors work is not required):

```
00011010011101010...
```

where each position represents a job title to a data base decode table. In other words, the system knows that the first position represents asphalt burner, the second candle snuffer, etc. Having gotten the list, you might then see how long it actually is with the CB function, for example

```
#(CP:#(CB:LIST):10:*040.00:*050.00:*060.00)
```

which says compare the count of bits in string LIST with 10, and if it is less than 10 go to statement 040.00, if it is equal ot 10 go to statement 050.00, and if it is greater than 10 go to statement 060.00.

At statement 040.00, you might want to simply print the list:

```
#(DC:OCCUPATIONS:WORK OUTDOORS:TEMP2)
#(PS:#(TEMP2))
```

but at statement 060.00 you might have the routine

```
060.00  #(PS:THERE ARE #(CB:TEMP) JOBS ON YOUR LIST.  WOULD YOU LIKE TO
        SEETHE FIRST PART OF THIS LIST?
061.00  KEYBOARD
062.00  DECOMP:(POS):064.00
063.00  GOTO 100.00
064.00  #(DC:OCCUPATIONS:WORK OUTDOORS:TEMP2))
        #(PS:#(TEMP2))
        etc.
```

Perhaps, as in the preference scripts, it would be useful to combine several of these lists from the data base and get the names of all the things that appear on all of the lists. For this we have to use Boolean logic, but in a very simple way.

There are four functions we need for combining Boolean lists. These are "and", "or", "exclusive or", and "not".

"AND" is the function that determines those items which are on each of two lists. In TRAC it is

```
#(BA:N1:N2)
```

which says, compare all the items in list N1 and list N2 and put all those items that are on both lists into list N1. This looks very much like the "compare for equalities" TRAC function, but there is one very basic difference: "Compare for equalities" operates on a list of English words, whereas "Boolean and" operates on a binary code that must be interpreted by a decode table to find the English words it represents. This is also true of the rest of the Boolean functions.

"Boolean or" determines all those things that are either in list N1 or in list N2. "Boolean exclusive or" determines those things in list N1 that do not appear on list N2 and those things in list N2 that do not appear in list N1. For example, suppose we had two lists that were coded representations of

LIST1: A B C D E M N O P

and LIST2: M N O P X Y Z

If we applied the Boolean "and" function, we would have the list

LIST3: M N O P

as a result. If we applied the Boolean "or", we would have

LIST4: A B C D E M N O P X Y Z

and the Boolean "exclusive or" function would return the list

LIST5: A B C D E X Y Z

The Boolean "not" function gives us a way to invert the sense of a list. For example, if the data base contained a list of occupations requiring union membership, and you were interested in a list of jobs that did NOT require union membership, you would apply the Boolean "not" function to the first list, and have the second as a result.

In order to get these Boolean coded lists from the data base, we use the function: FO

```
#(FO:OCCUPATIONS:UNION:LIST:*003.00)
```

This puts the coded information in string LIST.

In order to decode a Boolean coded string, once it has been operated on, we use the function:

```
#(DC:FILE:TABLE:N)
```

See HASM/DASM write-up for data available on ISVD system.

5. COMMUNICATION BETWEEN SCRIPTS

There are two main linkage strings in the system, LINK and STOP.

A return link, or a STOP return point, is set up with the "new top" TRAC function:

```
             #(NT:LINK:THISSCRIPT 006.00)
or           #(NT:STOP:THISSCRIPT 007.00)
```

A script wishing to do a "return", uses this instruction:

```
             #(XQ:#(PT:LINK):#(PT:LINK))
```

which is, in this case, the same as

```
             #(XQ:THISSCRIPT:006.00)
```

which says, send control to (execute) script THISSCRIPT, statement 006.00

If the script is to branch off to another one, never coming back, the instruction to use is

```
             #(XQ:THATSCRIPT)
```

to start at the beginning, or

```
             #(XQ:THATSCRIPT:005.00)
```

to start at statement 5.00

If the script is to branch off to another one, expecting to return, the NT function should be used before the XQ, to set the return point.

Scripts that define strings should eliminate these from core before transferring, unless these strings are to be used by the next script. This is because space in core is rather limited, and only that material that is being used should be kept there.

6. PERMANENT FORMS

Because of space limitations in the BTSS swapping area, we put some commonly used forms in resident storage, where one copy served all users. Although this saves space in user core, these permanent forms are restricted in that they may not be redefined or deleted and the forms pointer cannot be moved. These forms are described in Appendix D. See comments in section III of this report.

7. A DYNAMIC DESCRIPTION OF TRAC CORE STORAGE FOR PROGRAMMERS

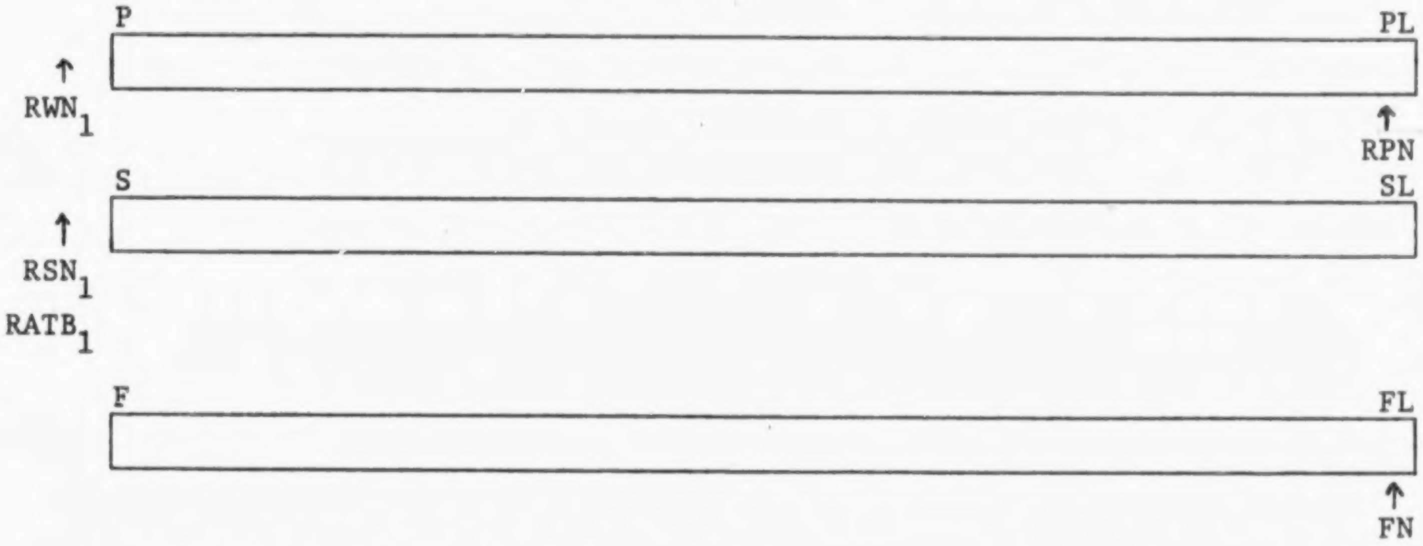
For those using the existing program, the GLURP listing itself and its flow charts explain what goes on in detail. For those who will be re-coding the program, a description of the TRAC storage management scheme as developed by Mooers and Deutch and others will be of interest. I found it rather neat compared to other systems which rely on myraid pointers to pieces of things and periodic "garbage collections" to retrieve space in core.

Let's return to the TRAC procedure

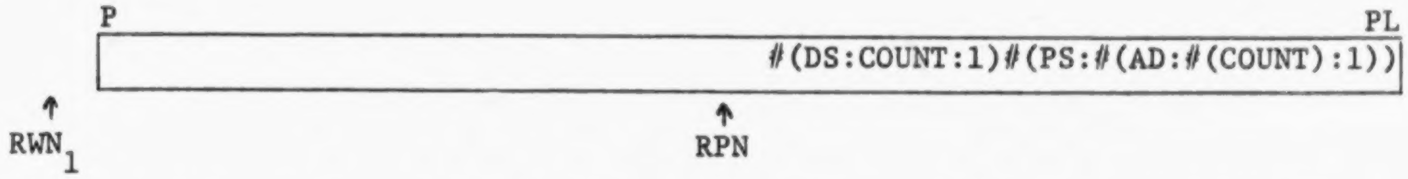
#(DS:COUNT:1)#(PS:#(AD:#(COUNT):1))

and trace the position of things in storage as it is interpreted. The areas we are interested in are the P, or active/neutral buffer, the F, or forms, buffer, and the S stack. PL, FL and SL mark the ends of these buffers, while registers RWN, RPN, FN and RSN point to the current position being referred to in these stacks. In TRAC convention, the pointer to the end of a string or a stack of other pointers is also used to point to the beginning of the next string or stack of pointers.

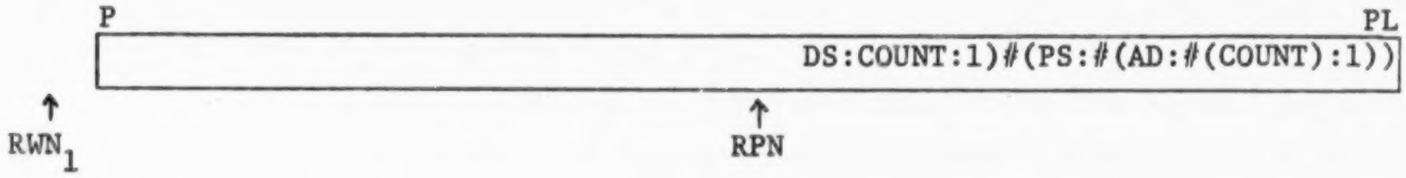
In the beginning, then, the buffers are empty with pointers pointing as shown:

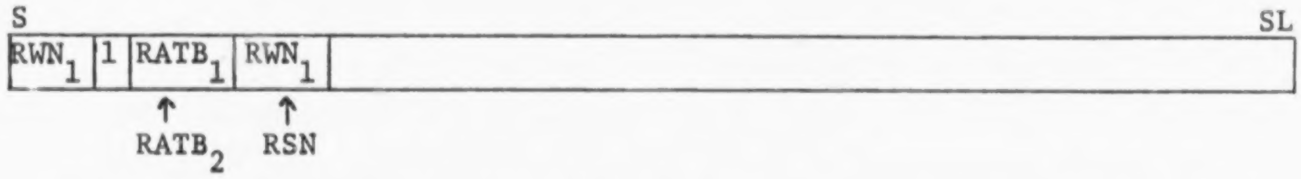


Routine INI discovers a TRAC type GLURP statement, moves it into the active stack, and adjusts RPN:



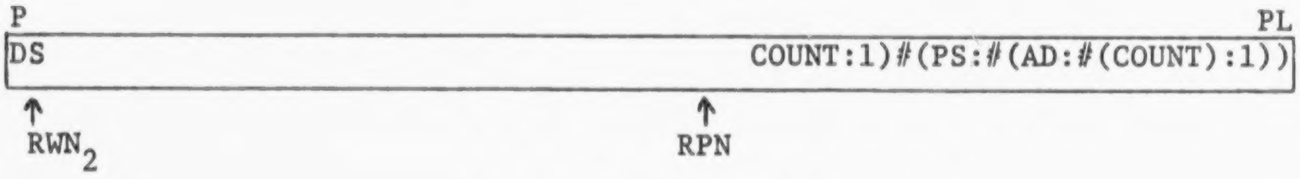
The interpreter now begins its scan at the position indicated by RPN, putting non-syntactic information in the neutral stack and pointing to the removed punctuation with pointers in the S stack. First it finds the "#(and marks it in S:

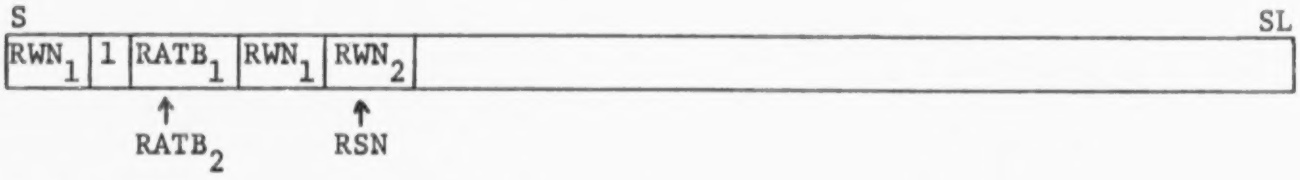



S
SL
RWN$_1$
1
RATB$_1$
RWN$_1$
RATB$_2$
RSN

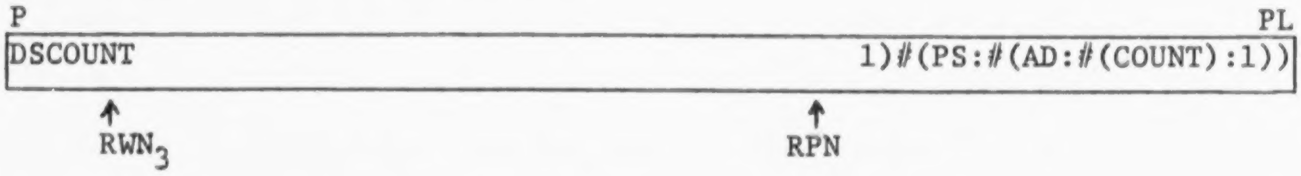
The first three S entries are for resuming scan after the function just found is evaluated. Entry 1 marks the place in the neutral stack to which the value of the up-coming function should be moved, or where the neutral stack marker should be if there is no value to the up-coming function. Entry 2 indicates that the deferred function is active when the scan resumes for that function, and entry 3 points to the beginning of the stack of pointers for the deferred function. These become more useful in the case of nested functions, as we shall see later. RATB is now set to its new beginning-of-function pointers position and one more entry is made to indicate the begin-argument-1 pointer of the current function.

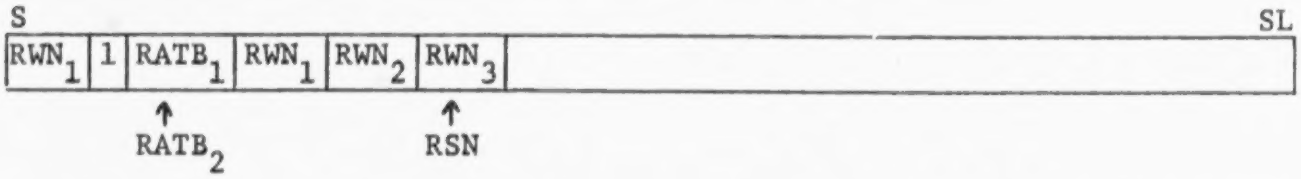
Scan continues to the colon, with non-syntactic characters going into the neutral stack and the colon noted in the S stack:


P
PL
DS
COUNT:1)#(PS:#(AD:#(COUNT):1))
RWN$_2$
RPN

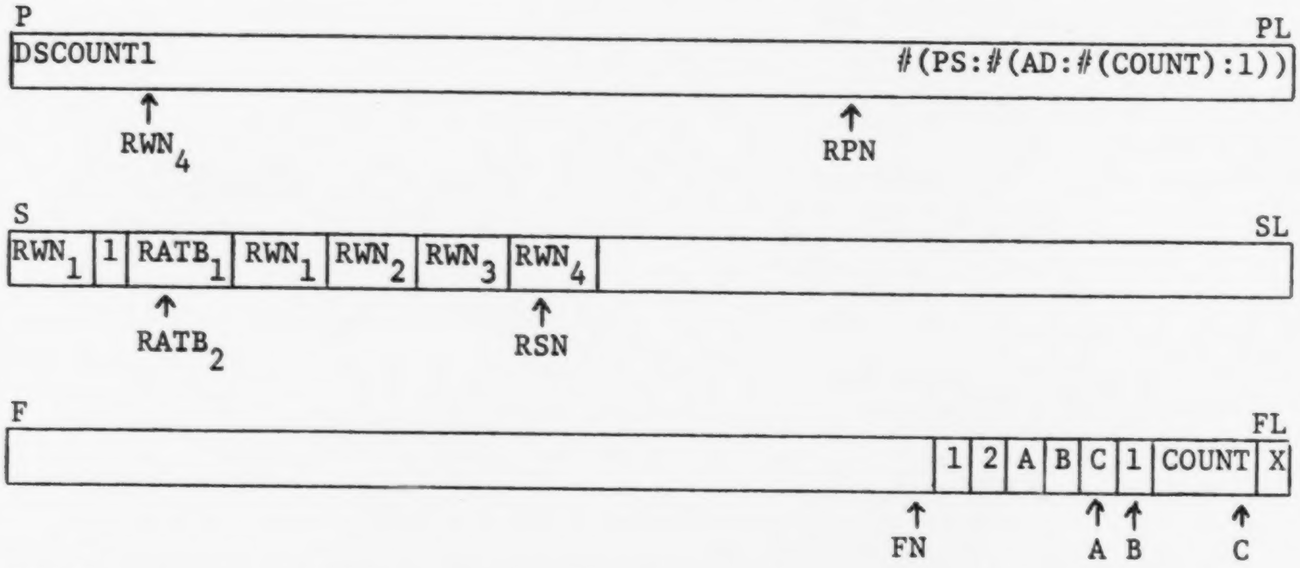

S
SL
RWN$_1$
1
RATB$_1$
RWN$_1$
RWN$_2$
RATB$_2$
RSN

And again:

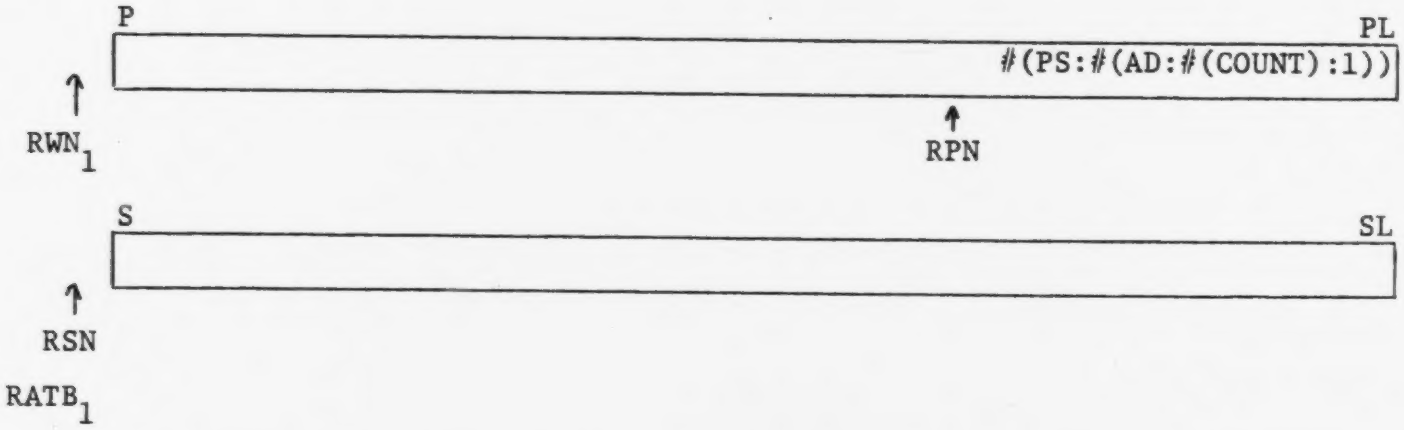

P
PL
DSCOUNT
1)#(PS:#(AD:#(COUNT):1))
RWN$_3$
RPN


S
SL
RWN$_1$
1
RATB$_1$
RWN$_1$
RWN$_2$
RWN$_3$
RATB$_2$
RSN

Now we find a right paren, mark it in the S stack and go off to execute the coding for primitive DS, which creates form COUNT is forms storage. The arguments for Ds are marked by the entries in S stack, argument 1 being from RWN_1 to RWN_2, argument 2 being from RWN_2 to RWN_3 and argument 3 being from RWN_3 to RWN_4:

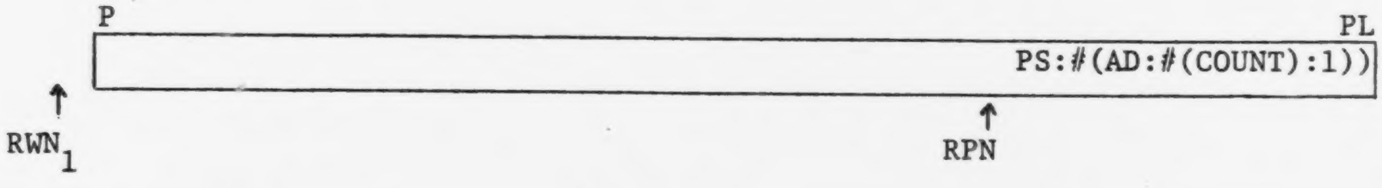

P
PL
DSCOUNT1
#(PS:#(AD:#(COUNT):1))
RWN_4
RPN
S
SL
RWN_1
1
$RATB_1$
RWN_1
RWN_2
RWN_3
RWN_4
$RATB_2$
RSN
F
FL
1
2
A
B
C
1
COUNT
X
FN
A
B
C

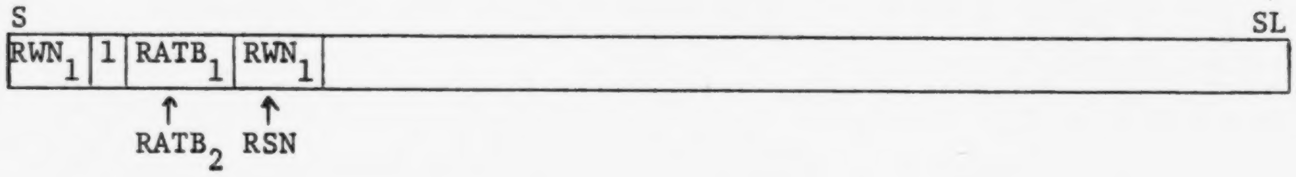
FUNDS returns to NULTRN, indicating that the function has no replacement value. Whereupon the interpreter recovers the space in the neutral stack (referring to entry 1), removes its notes in S stack about the function just finished, and picks up the rest of the pointers in the S stack to resume scan of the active stack:


P
PL
#(PS:#(AD:#(COUNT):1))
RWN_1
RPN
S
SL
RSN
$RATB_1$

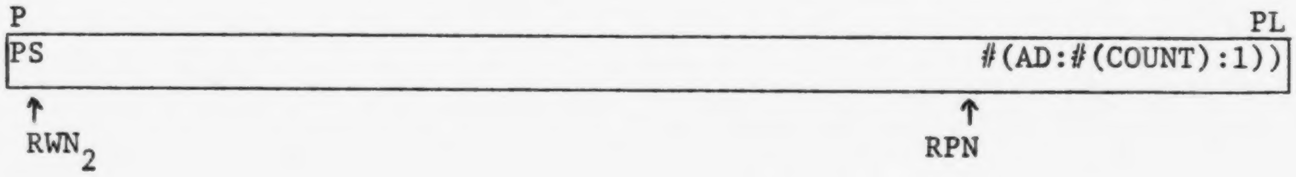
In the same fashion we analyze the rest of the functions in the active stack.

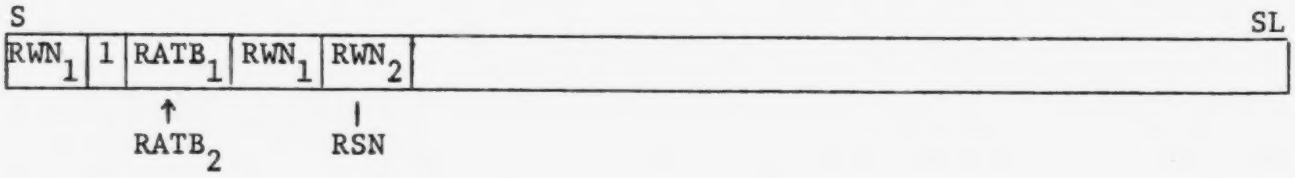
Step 1:


P
PL
PS:#(AD:#(COUNT):1))
RWN_1
RPN

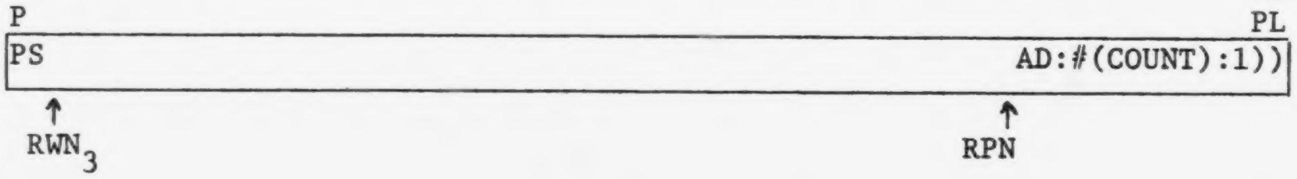

S
SL
RWN1
1
RATB1
RWN1
RATB2
RSN

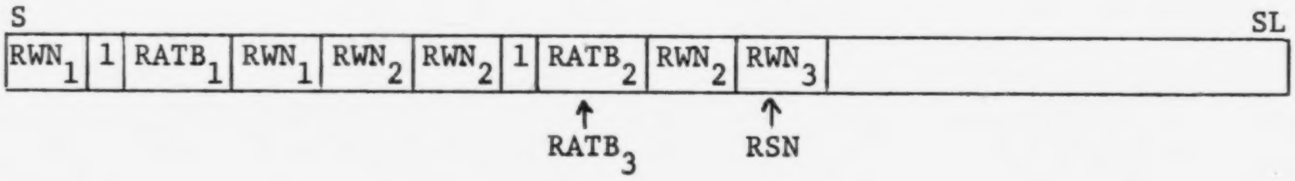
Step 2:


P
PL
PS
#(AD:#(COUNT):1))
RWN2
RPN

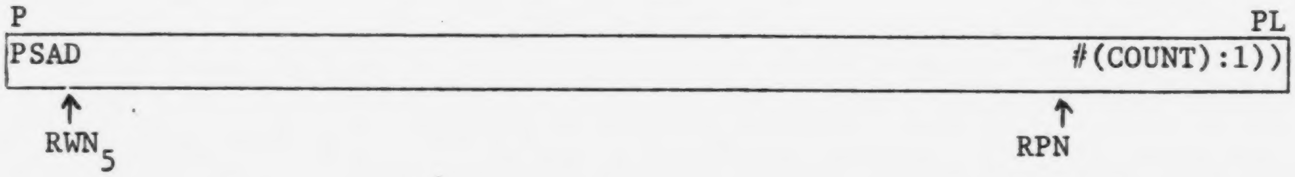

S
SL
RWN1
1
RATB1
RWN1
RWN2
RATB2
RSN

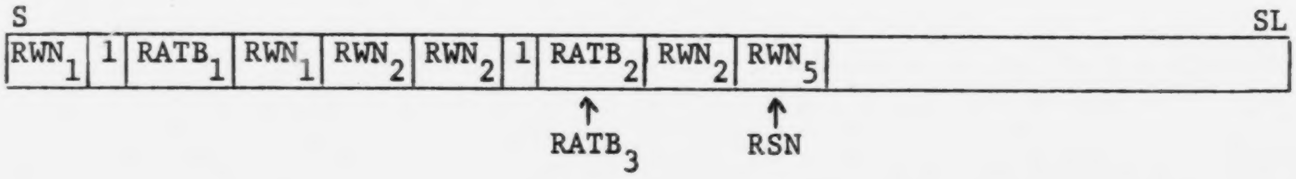
Step 3:


P
PL
PS
AD:#(COUNT):1))
RWN3
RPN

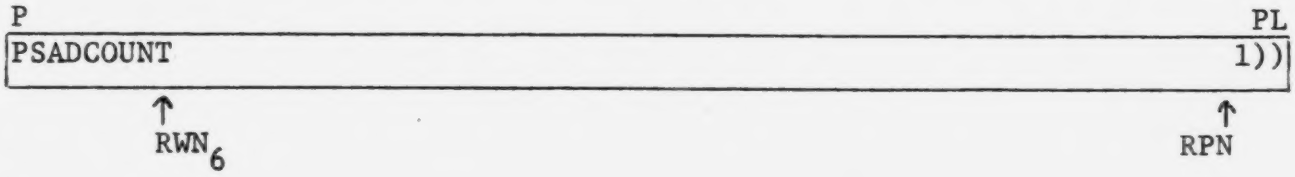

S
SL
RWN1
1
RATB1
RWN1
RWN2
RWN2
1
RATB2
RWN2
RWN3
RATB3
RSN

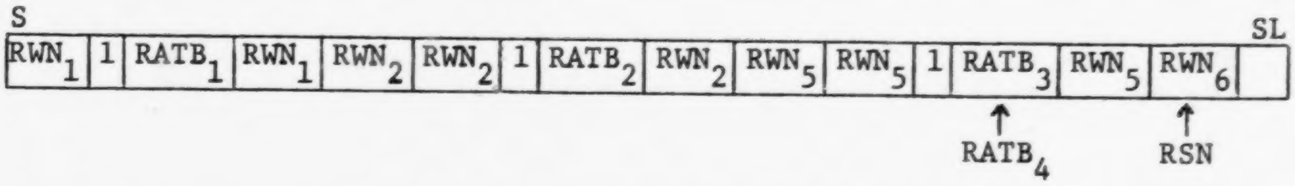
Step 4:


P
PL
PSAD
#(COUNT):1))
RWN5
RPN

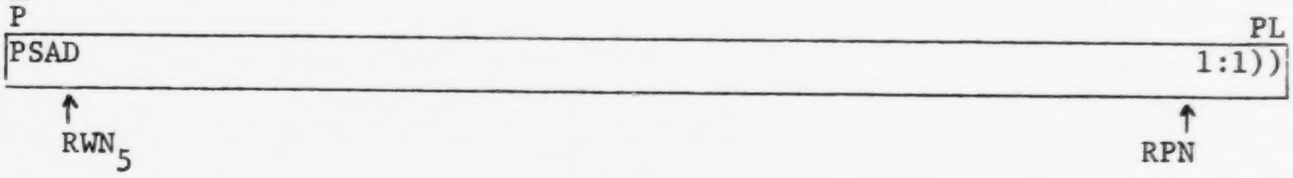

S
SL
RWN1
1
RATB1
RWN1
RWN2
RWN2
1
RATB2
RWN2
RWN5
RATB3
RSN

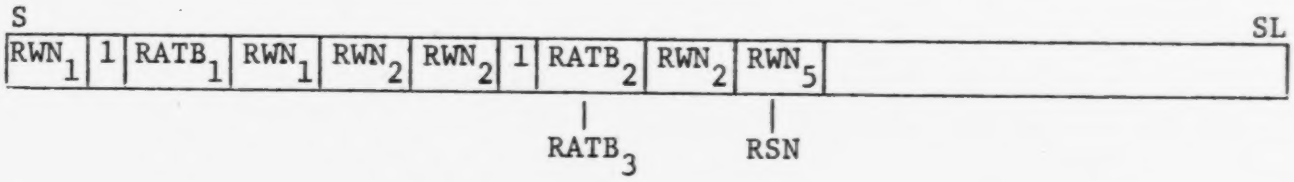
Step 5 and 6:

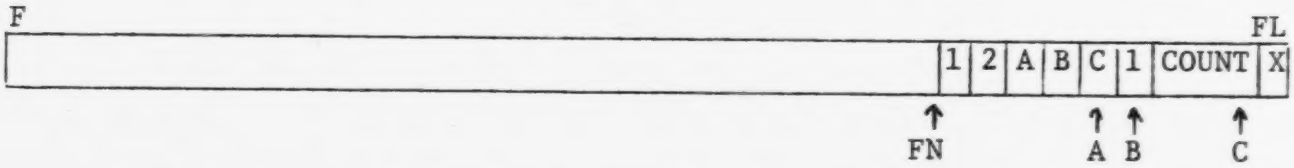

P
PL
PSADCOUNT
1))
RWN6
RPN


S
SL
RWN$_1$ | 1 | RATB$_1$ | RWN$_1$ | RWN$_2$ | RWN$_2$ | 1 | RATB$_2$ | RWN$_2$ | RWN$_5$ | RWN$_5$ | 1 | RATB$_3$ | RWN$_5$ | RWN$_6$
RATB$_4$
RSN

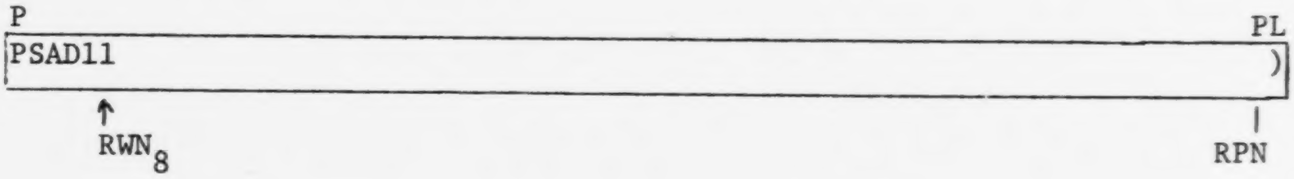
Here we finally find a right paren, which sends us off to execute the function. #(COUNT) is an implied call of form COUNT and has a replacement value of the contents of COUNT. Upon return from the function storage looks like this:

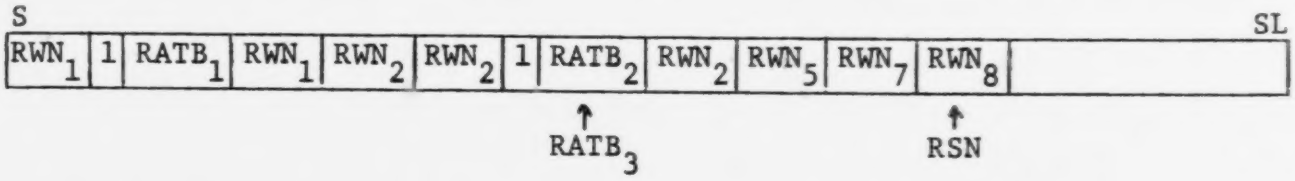

P
PL
PSAD
1:1))
RWN$_5$
RPN


S
SL
RWN$_1$ | 1 | RATB$_1$ | RWN$_1$ | RWN$_2$ | RWN$_2$ | 1 | RATB$_2$ | RWN$_2$ | RWN$_5$
RATB$_3$
RSN

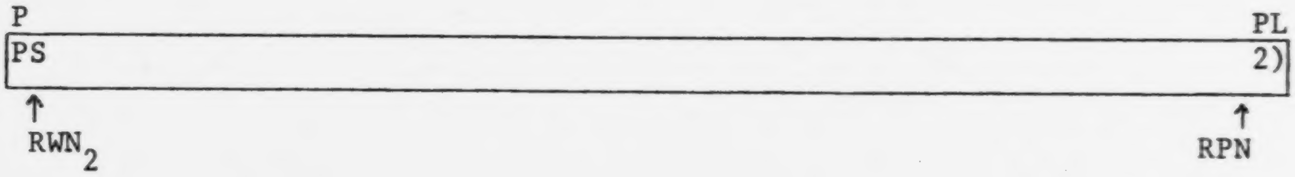

F
FL
1 | 2 | A | B | C | 1 | COUNT | X
FN
A
B
C

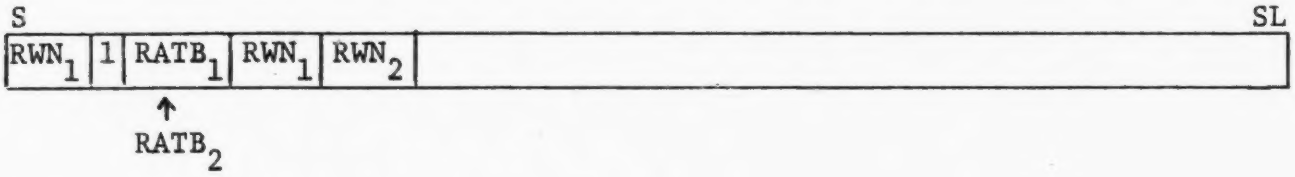
Note that the value of the function #(COUNT) has been put in the active stack for rescan, since the function was a single sharp, or active one. As we get to the next right paren the picture is:


P
PL
PSAD11
)
RWN$_8$
RPN

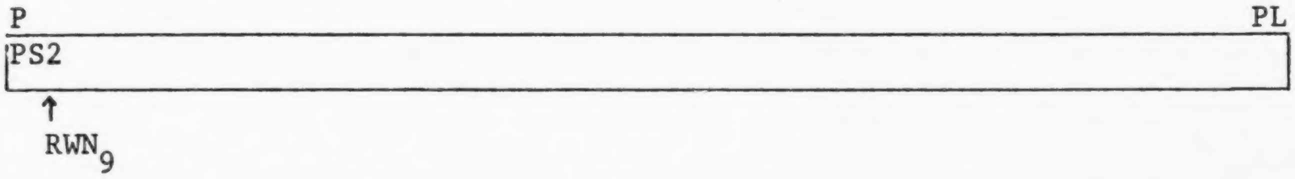

S
SL
RWN$_1$ | 1 | RATB$_1$ | RWN$_1$ | RWN$_2$ | RWN$_2$ | 1 | RATB$_2$ | RWN$_2$ | RWN$_5$ | RWN$_7$ | RWN$_8$
RATB$_3$
RSN

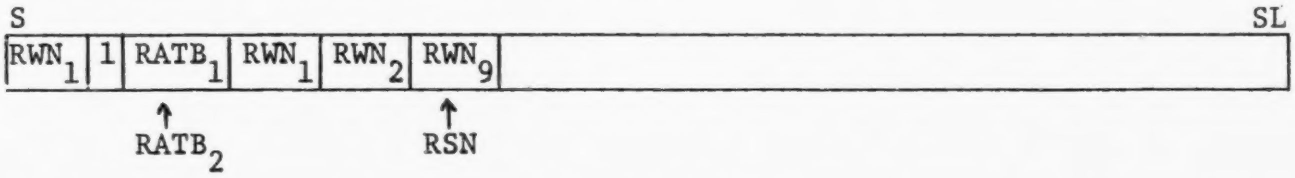
which evolves into:


P
PL
PS
2)
RWN$_2$
RPN



and:





"2" appears on the console, the end of the active stack is found, and control returns to INI for the next step.

6. AUTOMATIC LOGGING ROUTINES

In order to help script authors to evaluate their scripts . . . where they are misinterpreted by students, where they fail to respond appropriately to students, how they are most often used, etc. . . . we made use of the existing BTSS logging mechanism to record certain events on the "log tape." These entries are made to the log as they occur, with identification of user, time of occurrence, etc., associated with each message. See write-ups of BTSS log routines (RCA provided material) and the log print routines for more detail.

The call to log a message requires the following procedure: the 2-character s-code must be in location SCODE, the address of the log routine to format the proper message must be in location CODEGG, and control should be transferred to the address specified in SETLOGAD with that address in register 1. The log routines themselves run in P2 so that P1 registers remain undistrubed upon return from a log call. See write-up of ISVD log routine for further information.

II. ADAPTING GLURP TO OTHER BAL-ORIENTED MACHINES

1. GLURP INTERFACE WITH THE BTSS SYSTEM

Communication with the BTSS system from a Pl program is through SVC's. GLURP uses seven of these, which must be either deleted or replaced in the transfer to another system.

a. GET and the KEYBOARD routine

Used by GLURP to access material from the console. (We also made changes to the BTSS console I/O routine itself to fit the Sanders 720 console, but I'll assume that the receiving system already communicates with the consoles to be used.)

GET is used in the KEYBOARD routine, preceded by a PUT with the get bit in RFLAG set to tell the system a GET follows, and followed by a DELETEF to erase the message from the system buffers. (see FAPMAP write-ups.) This routine is highly specific to the Sanders 720 and would have to be rewritten for another console.

The KEYBOARD routine includes two other functions besides receiving information from the console. It makes note of a possible HELP branch specification, which it stores in HELPPT, and it sets up pointers in PBUFF, pointing to each word of the newly received input.

Also note that the LK function uses the KEYBOARD routine to move a message from core storage into the input buffers and set up the pointers in PBUFF.

If the student types a TRAC statement at KEYBOARD or there has been an error, entry point KEY2 is used to repeat the GET.

b. PUT and the CRT routines

Used by GLURP to send information to the console. PUT is used in the OUT2 routine, and when entered carries in register 1 the address of the calling sequence at BUFF3 and described in detail in the FAPMAP write-up. (RCA provided material.)

The various CRT routines and the OUT routines do considerable formatting of the output string to make it acceptable to the Sanders 720 and would have to be re-written for another system and/or console. Note that OUT2 is merely an extension of the CRT coding and that the CRT coding follows the specifications as stated in the language description (section I). PUT is also used to tell the BTSS system to be ready to receive from the console just before a GET.

c. DELETEF

Used in conjunction with GET to tell the BTSS system we are finished with the first record in the input stream and that it should be deleted from the buffers.

d. CEOT

In theory, this checks to see if the student has pushed a "break" or "interrupt" key, in which case we start over again.

e. EOT

Used to transfer control from GLURP to the BTSS system.

f. LOG

Used to send material to the BTSS log routine.

g. SETP2

Used to set GLURP into the P2 programming mode. This was required because there was no room in the user area to build up a log message.

2. GLURP INTERFACE WITH THE DISC

As it exists, GLURP relies on routine HASM to retrieve information from the disc. In designing the system, we assumed that the most experimental part of the software would be the disc storage and retrieval system. Therefore, in order to insure that scripts would not have to be rewritten to reflect each new data storage scheme, data access would remain symbolic at the script and GLURP levels. HASM is the routine, then, that would be re-written as the interface between GLURP and whatever data access routines were available on the new system.

Unfortunately, the symbolic aspect was not carried out fully. There are numerous TRAC functions related to the disc which may have to be redefined and rewritten or deleted in transforming GLURP into a new system. This is a simple procedure, as described in the next section.

3. CHANGING OR ADDING TO THE GLURP PROCESSOR

Control comes to the coding for a TRAC primitive with register 15 set as the base register to that coding. The arguments for the function have been completely parsed into the neutral stack with syntax pointers in the S stack. (see section I.7) SN holds the pointer to the last meaningful pointer in the S stack, which is the last character of the last argument of the function. If a function has a variable number of arguments, SN indicates the position of the last argument given in this particular function call. Otherwise, routines GET, GETF, and GOZ may be used to retrieve arguments. For descriptions of these and other possibly useful functions, see listing, "TRAC SUBROUTINES" section.

To add a primitive to GLURP, insert the 2-character mnemonic, in alphabetic order, in the TESTOR table, and the address to the coding for that function in the corresponding position of the TESTOR2 table. The actual coding for the primitive should be inserted in the "PRIMITIVES" area of core, for base register considerations.

To add a non-TRAC statement, insert the first 4 characters of the statement name, in alphabetic order, in the UNTABL, with the address to the coding for that function immediately following. The actual coding for the statement interpretation should also be in the range of the EASY base register.

4. STORAGE ASPECTS OF GLURP

The ISVD system is a dedicated one: its whole computing capacity is given over to the execution of the script network. This was done mainly because we were working from the only then-existing Spectra time-sharing system which was dedicated to FORTRAN. It was practical, then, to make one re-entrant copy of the GLURP processor, residing in permanent, read-only (storage protected) core where FORTRAN had formerly resided. Its variables were located, as they had been in FORTRAN, in a specific user storage area. In BTSS this user area is a constant one, the material in it being swapped in and out as each user's time slot comes around. Other systems will have other ways to manage user areas, but by manipulating register 13, which is the base register to the storage area, GLURP may be used in a dedicated system, with multiple programming or time sharing, or as a user program in either mode (or combination mode).

The permanent forms were created in desperation during a space squeeze, and should be returned to a variable user area if possible. These, of course, then become part of a script and are modified with the script as new networks are developed. As it stands, the permanent forms are modified to reflect changes in storage of permanent forms.

5. SCRIPT DEPENDENCY IN GLURP

As it stands, GLURP insists that the first script accessed be LOGIN (coding at TRACLO), that a script retrieval error be processed by the STOP script at step 473.20 (coding at INI and locations STENTRY, STCONST) and that there exist scripts STOP, DATA, SUMMARY, and QUIT. The later requirement can be changed by changing the UNTABL and its associated coding in the "UN-NESTED FUNCTIONS" section.

III. SOME OF THE THINGS THAT ARE WRONG WITH GLURP

1. SYSTEM ASPECTS

The ISVD system resides either in the wrong software system or on the wrong machine or both. Had we known the time and effort required to make BTSS behave the way we wanted it to, and our inability to really make it work the way or thought it should, we probably would have started right out to write a system from scratch. This is especially true since we were finally limited to six consoles, which could just as easily have been handled in multi-programming mode without the inefficiencies of core swapping.

The original rationalization for a big machine was that a) we would be handling a very large number of consoles, and b) that we would be applying large statistical programs to the available data. Neither of these things can be evolved from the system as it stands.

If we were to start over again, I would suggest that the ISVD system be evolved as a user program for an existing time-sharing system, allowing the system to be used by the schools for several functions at once. I would also suggest that the data be made available not only to scripts by way of GLURP, but to FORTRAN or other computational oriented languages for use by researchers.

Mooers has suggested that TRAC could serve as the controlling language in a multi-purpose time-sharing environment. This may be true, but developing a system in this way is certainly beyond the scope of an educational system development project in view of the many usable existing time-sharing programs.

2. LANGUAGE ASPECTS

We chose TRAC as the basic element of our language because, among other things, it is interpretive and retains a great deal of flexibility that compiled code cannot provide. Also, TRAC is easier to use and to expand than other string manipulating languages.

The consequences of the language remaining interpretive rather than compiled, however, produces the decided disadvantage of being slow and space-consuming. We were not able, in the time given, to find a midpoint between the interpretive and compiled modes in the development of GLURP.

What we did instead was to create a second, completely separate mode for some statements that ran interpretively, but in a semi-fixed format and without macro-expanding capabilities. This added efficiency to GLURP, in that simple kinds of tasks (CRT, KEYBOARD) could be performed without lengthy analysis of what was to be done, but does not contribute to be homogeneous, flexible language.

It became obvious that the answer to all this is that the author language must be defined with the expectation that it will be at least pre-processed before execution if not compiled into machine language. Every attempt to achieve this, however, managed to reduce the flexibility given by TRAC. (see Appendix E)

Another obvious defect in GLURP is that it has no computational capabilities. Obviously, given a large data base, it would be nice to be able to apply statistical techniques to it. In the existing system not only does GLURP not lend itself to the handling of arrays, but the HASM/DASM routines do not lend themselves to data scanning and/or selective extracting of data.

We did some work in applying BEATON's work to TRAC, but set this aside -- permanently it turned out -- while we solved other problems. (see Appendix F) Had we had space enough, we would have left FORTRAN in the system and created a GLURP statement to call it in. The problem would still have remained, however, of how to built an array in TRAC to communicate to FORTRAN-like programs and/or how to make TRAC-like or FORTRAN-like programs able to scan the data base for specific classes of data. (see Appendix G)

Another defect in GLURP is its complexity from the author's standpoint. The whole matter of translating what the student wants into a data access command is extremely tedious and incomplete. The ELIZA method of determining keywords and subject matter, the English-like data interface, and the ability of TRAC to create strings from student input are all important to student control of the system. The author, however, is expected to anticipate, at every KEYBOARD statement, an incredible variety of possible inputs and to provide responses for them. The student command idea was introduced largely to help correct this problem . . . it was just too much to ask an author to check each time whether the student was trying to change the subject or not. The permanent rule lists (POS, NEG, etc.) were another attempt in this direction.

I think, given time, we would have looked harder at the ELIZA scheme of script levels, looking for ways that scripts could build on one another. DECOMP statements would, ideally, be built dynamically according to the previous context of student input, and probably in conjunction with a thesaurus stored on the disc. This means, of course, that the DECOMP statement would not look much at all like its present form. Linking and clean-up problems should be more automatic, allowing the author to concentrate on subject material rather than TRAC techniques.

In fact, the whole concept of keeping track of what the student had said, his interests, his choice of vocabulary, of what data he had actually accessed and what more, therefore he might be interested in, was left entirely up to the author. This was not our original intention. We had hoped to find a scheme whereby a context grid of some sort would not only be automatically made for the author, but would also be used by the data access routine to anticipate the needs of the student, thereby speeding up response time of the system.

At one point I played with the idea of a script language following the Tiedeman paradigm, whereby a block in the script would have a format somewhat like this:

1. text presentation
2. accept student input
3. GLURP decides whether input is a statement or a question. If it's a question, go to question-answering script and return to step 1. We keep track here of whether the student has put himself in "exploration" mode, and what pieces of data he is requesting.
4. Analysis by author of "correctness" or context of input and branch appropriately.

This led to the idea of the HELP branch on the KEYBOARD statement, and student commands, but to nothing in the way of "monitoring" or automatic record-keeping.

The ELIZA language was found to be cumbersome in its lack of concept of sequential steps within a script. GLURP, on the other hand, in being primarily sequential in flow, does not encourage the script writer to think in terms of student input rather than author output. Somewhere in between lies the ideal.

Certain critics have told us the ISVD system should have been coded in FORTRAN so that it could be transferred easily to another machine. Aside from the obvious lack of familiarity with system implementation behind this sort of statement, I am glad that we did not do so. I don't think anyone on the project would consider the current system useful in the real world except for the lessons we have learned, the advancement we did make, and the hope it extends that such a system could be a great contribution to education and the presentation of data.

We have spent some time working on the problems of understanding English. More time should be spent considering how to make data available to students in more ways, and how to make life easier for the author in terms of automatic record-keeping.

APPENDIX A

TRAC, A Procedure-Describing Language for the Reactive Typewriter

Calvin N. Mooers
Rockford Research Institute, Inc.

included with permission of the author

TRAC, A Procedure-Describing Language for the Reactive Typewriter

Calvin N. Mooers

Rockford Research Institute Inc., Cambridge, Massachussetts

A description of the TRAC (Text Reckoning And Compiling) language and processing algorithm is given. The TRAC language was developed as the basis of a software package for the reactive typewriter. In the TRAC language, one can write procedures for accepting, naming and storing any character string from the typewriter; for modifying any string in any way; for treating any string at any time as an executable procedure, or as a name, or as text; and for printing out any string. The TRAC language is based upon an extension and generalization to character strings of the programming concept of the "macro." Through the ability of TRAC to accept and store definitions of procedures, the capabilities of the language can be indefinitely extended. TRAC can handle iterative and recursive procedures, and can deal with character strings, integers and Boolean vector variables.

Introduction

The TRAC (Text Reckoning And Compiling) language system is a user language for control of the computer and storage parts of a reactive typewriter system. A reactive typewriter is understood to be one of a number of teletypewriters simultaneously connected online by wire to a memory and computer complex which permits real-time, multiple access (time-shared) operation. In the philosophy of the reactive typewriter, the man at the typewriter keyboard is the focal point of the system. The connected storage and computer devices are considered to be peripheral service units to the reactive typewriter.

The design goals for the TRAC language and its translating system included: (1) high capability in dealing with back-and-forth communication between a man at a keyboard and his work on the machine, so as to permit him to make insertions and interventions during the running of his work; (2) maximum versatility in the definition and performance of any well-defined procedure on text; (3) ability to define, store and subsequently use such procedures to extend the capabilities of the language; and finally (4) maximum clarity in the language itself so that it could be easily taught to others. A discussion of these design goals, and of the design decisions which went into

---

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965; a supplementary paper, not on the announced program.

The present work was supported in part by the following grants and contracts: Advanced Research Projects Agency contract SD-295, Information Sciences Branch of the Air Force Office of Scientific Research contracts AF-AFOSR 376, 377 and 461-64, and Division of General Medicine, National Institutes of Health, U.S. Public Health Service grant GM 10416.

the language, may be found in a companion paper [1]. The Trac language has now been programmed for several computers.[1] It has shown a high degree of stability during the past year of experimental use.

The present Trac language is machine-independent and is closed with respect to operations performed upon sets of characters from a typewriter keyboard. The present language should be precisely designated as "Trac 64."[2] Later versions of Trac are expected to have the ability to deal with strings of machine-coded words and with subroutines, and will thus be self-implementing.

The Trac language was developed after a study of a number of procedure-describing languages, but only after it was concluded that each of these had features which were believed to be unsuitable or unduly constraining for the purposes contemplated by Trac. In particular, the languages IPL-V, Lisp and Comit were carefully examined. In brief, IPL-V appeared to be too closely oriented to computer programming. Lisp had severe restrictions due to its "atomic" symbols, certain conceptual confusions and too great an orientation to mathematical logic. Comit, while suitable in many respects, had the rigidity associated with a compiler. When work was begun on Trac (1960) none appeared to have the capabilities desired, though they were a definite source of inspiration.

The prime stimulus to the present Trac language came from two important unpublished papers by Eastwood and McIlroy [3] and McIlroy [4]. The first paper described a macro assembly system having run-time definition-making and decision-making capabilities. The second paper showed how this system could perform very general manipulations on symbol strings. Trac is a refinement and extension of the macro approach of these papers. Therefore it can be said that the present Trac system consists of a machine-independent language together with a generalized macro text processor which runs interpretively to provide versatile interaction capabilities at run time. A recent, independently developed system by Strachey [5] has a number of remarkable similarities to Trac.

TRAC Syntax

A Trac string may contain a substring enclosed by a matching pair of parentheses, such as (···) where the dots indicate a string. The matching parentheses indicate the scope of some particular action. There are three cases, epitomized by #(···), ##(···) and (···).[3] The first two formats indicate the presence of a Trac "primitive function." The format #(···) denotes an "active function," while the format ##(···) denotes a "neutral function." This distinction is clarified below. The string interior to either kind of function is generally divided into substrings by commas as in #(, ,) where these substrings constitute the arguments of the function. Parentheses in the format (···) have roughly the same role as paired quotation marks, and, in particular, whatever string is inside the paired parentheses is protected from functional evaluation.

Trac strings are dealt with by the Trac processor according to a scanning algorithm which works from left to right and performs the evaluation of nested expressions from inside outward. In the expression

```
#( ,#( , , #( ) , ##( ) ))
4   3        1      2
```

the functions are evaluated in the order indicated. As each function is evaluated, it is replaced in the Trac string by the string (possibly null) which is its value. The evaluation of an active function is followed directly by the evaluation of any function in its value string not protected by matched parentheses. The value string of a neutral function is not further evaluated.

Examples of Functions

An example of the "define string" primitive function is the expression #(*ds*,*AA*,*CAT*). This causes recording of the string *CAT* in the memory and places the name *AA* of the string in a table of contents. The string can be called out of memory by the "call" function #(*cl*,*AA*). The result of the call function is to place the string value of the call, namely *CAT*, in the former location of #(*cl*,*AA*), with an expansion or a closing up of the surrounding strings. The call function is in the class of functions having a "value string." The define string function is an example of a function having a "null value"; i.e., no string is left behind in its place after its evaluation.

Evaluation of the "read string" function #(*rs*) causes the processor to accept input from the typewriter. Its value is the string as received from the typewriter up to a terminating "meta character" which is usually taken to be the apostrophe. The meta character can be changed. The "print string" function #(*ps*,*X*) causes printing out of the argument string, here represented by the symbol *X*. It has null value. The nested expression #(*ps*,#(*cl*,*AA*)) will cause *CAT* to be printed out.

In the beginning, and at the completion of every processing cycle, the Trac "idling procedure" #(*ps*,#(*rs*)) is automatically loaded into the Trac processor. It is therefore seen that all strings and programs are effectively loaded into the interior of the idling procedure, and furthermore, all Trac computations are made on functions nested within the argument string of some other function.

TRAC Algorithm

The Trac algorithm governs the precise manner in which Trac expressions are scanned and evaluated by the Trac processor. At the beginning, the unevaluated strings are in the "active string" and the "scanning pointer" points to the leftmost character in this string. As characters have been treated by the scanning algorithm, they may be added to the right-hand end of a "neutral string," which is so called because its characters have been fully treated by the algorithm and are thus neutral, like alphabetic characters. The algorithm follows.

---

[1] The computers are the Digital Equipment PDP-1, PDP-5, the General Electric Datanet-30 and the Scientific Data Systems SDS 930.

[2] The present paper is a revision and extension of [2].

[3] With the model 33 Teletype, the "up arrow" character is preferred instead of the sharp sign.

1. The character under the scanning pointer is examined. If there is no character left (active string empty), go to rule 14.

2. If the character just examined (by rule 1) is a begin parenthesis, the character is deleted and the pointer is moved ahead to the character following the first matching end parenthesis. The end parenthesis is deleted and all nondeleted characters passed over (including nested parentheses) are put into the neutral string without change. Go to rule 1.

3. If the character just examined is either a carriage return, a line feed or a tabulate, the character is deleted. Go to rule 15.

4. If the character just examined is a comma, it is deleted. The location following the right-hand character at the end of the neutral string, called the "current location," is marked by a pointer to indicate the end of an argument substring and the beginning of a new argument substring. Go to rule 15.

5. If the character is a sharp sign, the next character is inspected. If this is a begin parenthesis, the beginning of an active function is indicated. The sharp sign and begin parenthesis are deleted and the current location in the neutral string is marked to indicate the beginning of an active function and the beginning of an argument substring. The scanning pointer is moved to the character following the deleted parenthesis. Go to rule 1.

6. If the character is a sharp sign and the next character is also a sharp sign, the second-following character is inspected. If this is a begin parenthesis, the beginning of a neutral function is indicated. Two sharp signs and the begin parenthesis are deleted and the current location in the neutral string is marked to indicate the beginning of a neutral function and the beginning of an argument substring. The scanning pointer is moved to the character following the deleted parenthesis. Go to rule 1.

7. If the character is a sharp sign, but neither rule 5 or 6 applies, the character is added to the neutral string. Go to rule 15.

8. If the character is an end parenthesis, the character is deleted. The current location in the neutral string is marked by a pointer to indicate the end of an argument substring and the end of a function. The pointer to the beginning of the current function is now retrieved. The complete set of argument substrings for the function have now been defined. The action indicated for the function is performed. Go to rule 10.

9. If the character meets the test of none of the rules 2 through 8, transfer the character to the right-hand end of the neutral string and go to rule 15.

10. If the function has null value, go to rule 13.

11. If the function was an active function, the value string is inserted to the left of (preceding) the first unscanned character in the active string. The scanning pointer is reset so as to point to the location preceding the first character of the new value string. Go to rule 13.

12. If the function was a neutral function, the value string is inserted in the neutral string with its first character being put in the location pointed to by the current begin-of-function pointer. Delete the argument and function pointers back to the begin-of-function pointer. The scanning pointer is not reset. Go to rule 15.

13. Delete the argument and function pointers back to the begin-of-function pointer for the function just evaluated, resetting the current location to this point. Go to rule 15.

14. Delete the neutral string, initialize its pointers, reload a new copy of the idling procedure into the active string, reset the scanning pointer to the beginning of the idling procedure, and go to rule 1.

15. Move the scanning pointer ahead to the next character. Go to rule 1.

The TRAC processor will accept any string of symbols. Nonexistent functions are given a null value. Omitted arguments are given a null value, while extra arguments are ignored. Omitted right parentheses will cause the processor to terminate its action and reinitialize itself at an unexpected point, while extra right parentheses are ignored and deleted at the end of a procedure. When the processor becomes too full, perhaps due to an infinite iteration or recursion, a diagnostic is typed out to indicate that fact and the processor is re-initialized by going to rule 14. The break key stops any action and causes re-initialization.

The TRAC Functions

Input-Output. All functions are shown in their active representation, which is the form most often used. As shown, the argument strings are presumed not to contain functions or other active matter.

#(*rs*) "read string" (one argument). (Note that the mnemonic for the function name is counted as the first argument.) The value is the string as read from the teletypewriter keyboard up to the point of occurrence of the meta character, which is deleted.

#(*rc*) "read character" (one argument). The value is the next character, which may be any character (including the meta character) received from the teletypewriter.

#(*cm*,X) "change meta" (two arguments). This null-valued function changes the meta character to the first character of the string symbolized by X. Upon starting, the TRAC processor is loaded with a standard meta character, usually the apostrophe.

#(*ps*,X) "print string" (two arguments). This null-valued function prints out on the teletypewriter the string represented by X.

Define and Call Functions

#(*ds*,N,X) "define string" (three arguments). This is a null-valued function. The string symbolized by X is placed in storage and is given the name symbolized by N. The name is placed in a name list or table of contents to the "forms" in storage. A "form" is a named string in storage. If a form is already in storage with name N, this form is erased. The name N may be a null string.

#(*ss*,N,$X1$,$X2$,$\cdots$) "segment string" (three or more arguments). This is a null-valued function. The form named N is taken from storage and is scanned from left to right with respect to string $X1$. If a substring is found matching $X1$, the location of the match is marked. The matching substring is excluded from further action, thus creating a "segment gap." The rest of the form is scanned with respect to $X1$ to create any additional segment gaps. These segment gaps are all given the ordinal value one. The parts of the form not taken by segment gaps are now scanned with respect to string $X2$, with the creation of segment gaps of ordinal value two. This action is repeated with all of the remaining argument strings. At the end, the marked form, along with its pointers and ordinal identifiers for the segment gaps, is put back into storage with the name N. The untouched portions of the string in the form are called "segments." It is seen that the segment string function creates a "macro" in which the arguments $X1$, $X2$, etc., indicate the dummy variables. The segment string function subsequently can be applied with other arguments to the same form, with the result of new segment gaps being created with ordinal value one, two, etc., and being inserted among those already there. A null string for one of the arguments X causes no action for this argument.

#*(cl,N,X1,X2,···)* "call" (two or more arguments). The value is generated by bringing the form named N from storage and filling the segment gaps of ordinal value one with string $X1$, the gaps of ordinal value two with string $X2$, and so on for all the segment gaps in the form.

The following specialized calls read out a part of a form. They treat the segment gaps as if the gaps were filled with the null string. These calls (*cs*, *cc*, *cn*, and *in*) preserve the neutral-active function distinction only for the strings coming from the form named N. Since the alternative value of these functions, symbolized by Z, may be a call to a procedure, the alternative value is always treated as if the function were active.

All the call functions (*cl*, *cs*, *cc*, *cn*, and *in*) read the text of a form beginning at the location indicated by a "form pointer" which is part of the apparatus of the form. Initially the form pointer points at the first character of the form. The call function does not change the form pointer.

#*(cs,N,Z)* "call segment" (three arguments). The value of this function is the string from the current location of the form pointer to the next segment gap of the form named N. If the form is empty, the value is Z. The form pointer is moved to the first character following the segment gap.

#*(cc,N,Z)* "call character" (three arguments). The value is the character under the form pointer. If the form is empty, the value is Z. The form pointer is moved one character ahead (segment gaps are skipped).

#*(cn,N,D,Z)* "call *n* characters" (four arguments). This function reads from the form named N from the point indicated by the form pointer and continuing for a number of characters specified by the decimal integer number at the tail end of the string symbolized by D. Segment gaps are skipped. If the decimal number is positive, this function reads the string to the right of the pointer; if negative, to the left. The strings so read are preserved in their character sequence. If no characters are available to be read, the value is Z. The form pointer is moved (right or left) to the next unread character.

#*(in,N,X,Z)* "initial" (four arguments). Starting from the form pointer, the form named N is searched for the first location where the string X produces a match. The value is the string from the pointer up to the character just before the matching string. If a match is not found, the value is Z. The form pointer is moved to the character following the matching substring, or is not moved if there is no match.

#*(cr,N)* "call restore" (two arguments). This null-valued function restores the form pointer of the form named N to the initial character.

#*(dd,N1,N2,···)* "delete definition" (two or more arguments). This null-valued function deletes the forms named $N1$, $N2$, etc., from memory and removes their names from the list of names.

#*(da)* "delete all" (one argument). This null-valued function deletes all the forms in memory, and removes their names.

Arithmetic Functions. TRAC does integer arithmetic, taking decimal arguments. The decimal numeric digits are looked for at the tail ends of the argument strings. The prefix string of the first argument string is preserved and is appended to the answer, while the prefix string of the second argument is ignored. Negative quantities are indicated by the minus sign "−", and initial zeros are ignored. Whenever the integer values become so large as to overflow the capacity of the arithmetic processor, the overflow value Z of the function is taken. The overflow value is always treated as if it were produced by an active function. The arithmetic functions are: #*(ad,D1,D2,Z)* "add", #*(su,D1,D2,Z)* "subtract", #*(ml,D1,D2,Z)* "multiply" and #*(dv,D1,D2,Z)* "divide". They all take four arguments. In these functions, $D2$ is subtracted from $D1$, and $D1$ is divided by $D2$, with the answer being the largest integer contained in the dividend.

Boolean Functions. Boolean TRAC functions operate on strings of bits (of value 0 or 1), i.e., on Boolean vectors. The bit strings are represented by octal digits, with each digit representing three bits. Thus the bit strings have lengths in multiples of three. The octal digits are looked for at the tail end of the 01 and 02 strings, and any nonoctal prefix matter is deleted The functions are: #*(bu*,01,02) "Boolean union," #*(bi*,01,02) "Boolean intersection," #*(bc*,01) "Boolean complement," #*(bs,D1*,01) "Boolean shift" and #*(br,D1*,01) "Boolean rotate." The bit strings are right justified. In the Boolean union the shorter string is filled out with leading zeros, while in the Boolean intersection, the longer string is truncated at the left. In the complement, shift and rotate, the length of the bit string remains the same. Shift is to the left by the number of places specified by the decimal $D1$ (with leading nondecimal matter being deleted) when

D1 is positive, and to the right when D1 is negative. The new positions created by the shift are filled with zeros. Rotate is also to the left or right, with positive or negative values of D1. The digits displaced from one end of the vector are added to the place created at the other end.

Decision Functions

#(*eq*,X1,X2,X3,X4) "equals" (five arguments). This is a test for string equality. If X1 is equal to X2, the value is X3; otherwise it is X4.

#(*gr*,D1,D2,X1,X2) "greater than" (five arguments). This is a test of numerical magnitude. If the integer decimal number at the tail of string D1 is algebraically greater than the number at the tail of D2 the value is X1; otherwise it is X2.

External Storage Management Functions

#(*sb*,N,N1,N2,···) "store block" (three or more arguments). This null-valued function assembles the group of forms named N1, N2, etc., and stores them as a block in an external storage area. The form names, segment gaps, etc., are all preserved. When the forms have been put into the external storage, they are erased from form storage. A new form is created with name N and with a string which is the address of the block in external storage.

#(*fb*,N) "fetch block" (two arguments). This null-valued function is the converse of the store block function. The name N is the name of the block to be fetched. The function restores to form storage all the forms in the block, complete with names, segment gaps, pointers, etc. It does not erase the block in external storage, nor the form named N.

#(*eb*,N) "erase block" (two arguments). This null-valued function erases the form named N and also the group of forms in the block in external storage.

These functions permit forms to be moved to and from the main memory and also protect the stored forms from accidental erasure. They also permit one to build a "storage tree." By this technique, a group of forms can be stored under a group name, a set of group names can be stored under a section name, and so on.

Diagnostic Functions

#(*ln*,X) "list names" (two arguments). The value of this function is the list of names in the name list, i.e., the names of all the forms in form storage. Each name in the value string is preceded by string X. If X is the character pair "carriage return, line feed" protected by double parentheses, the names will be listed in a column.

#(*pf*,N) "print form" (two arguments). This causes the typing out of the form named N with a complete indication of the location and ordinal values of the segment gaps.

#(*tn*) "trace on" (one argument). This null-valued function initiates the trace mode in which, as the computation progresses, the neutral strings for each function are typed out. Typing the backspace key causes evaluation of the function, and presentation of the neutral strings of the next. Typing anything other than backspace causes initialization. Carriage return may be used instead of backspace.

#(*tf*) "trace off" (one argument). This is a null-valued function which terminates the trace mode without initialization. Both trace on and trace off functions may be placed anywhere in a procedure.

Examples of TRAC Procedures

1. The distinction between active and neutral functions is usually puzzling. In essence, the value from an active function is rescanned, while in the neutral function it is not. The following example shows the action of the protective parenthesis, the neutral and the active forms of the function. Consider that both #(*ds*,*AA*,*CAT*)′ and the simple program #(*ds*,*BB*,(#(*cl*,*AA*)))′ have been presented to the processor. Then,

```
#(ps,(#(cl,BB)))',     #(ps,##(cl,BB))',     #(ps,#(cl,BB))'
```

prints out, respectively:

```
#(cl,BB),              #(cl,AA),             CAT
```

2. When the processor is quite full, it is often desirable to delete all forms but one of a particular name. The procedure #(*ds*,*N*,##(*cl*,*N*)#(*da*)) will accomplish this. Here ##(*cl*,*N*) reads the form N into the processor, and it is held in the neutral string while all the forms in memory are erased. The form is then redefined with its original name. In this example, segment gaps are lost.

3. This and the following example illustrate the extension of TRAC capabilities through defining and storing of suitable procedures. The calculation of the factorial of a number can be done by simple recursion:

```
#(ds,Factorial,(#(eq,1,X,1,
(#(ml,X,#(cl,Factorial,#(ad,X,-1))))
)))#(ss,Factorial,X)'
```

Then the call #(cl,Factorial,5)′ produces the result 120.

4. Many users will prefer to have TRAC supply its own sharp signs and parentheses when calling a procedure. The following will do this:

```
#(ds,English,(#(ps,
#(cl,#(rs))(
))#(cl,English)))'
```

To start this action, we use #(*cl*,English)′, and then if one types in Factorial,5′ the response is 120 followed by carriage return, line feed. The action is terminated by #(*dd*,English)′.

Acknowledgments. Special thanks are due to my collaborator L. Peter Deutsch for his assistance in the development and implementation of TRAC.

REFERENCES

1. Mooers, C. N. TRAC, a text handling language. Proc. ACM 20th Nat. Conf. Cleveland, Aug. 1965, pp. 229–246.
2. TRAC—a procedure defining and executing system. Mem. V-157, Rockford Research, Cambridge, June 1964.
3. Eastwood, D. E., and McIlroy, M. D. Macro compiler modification of SAP. Mem., Comput. Lab., Bell Telephone Labs., Murray Hill, N.J., Sept. 3, 1959. (Unpublished)
4. McIlroy, M. D. Using SAP macro instructions to manipulate symbolic expressions. Mem., Comput. Lab., Bell Telephone Labs., Murray Hill, N.J., 1960. (Unpublished)
5. Strachey, C. A general purpose macrogenerator. *Comput. J.* 8, 3 (1966).

TRAC

A TEXT RECKONING AND COMPILING LANGUAGE

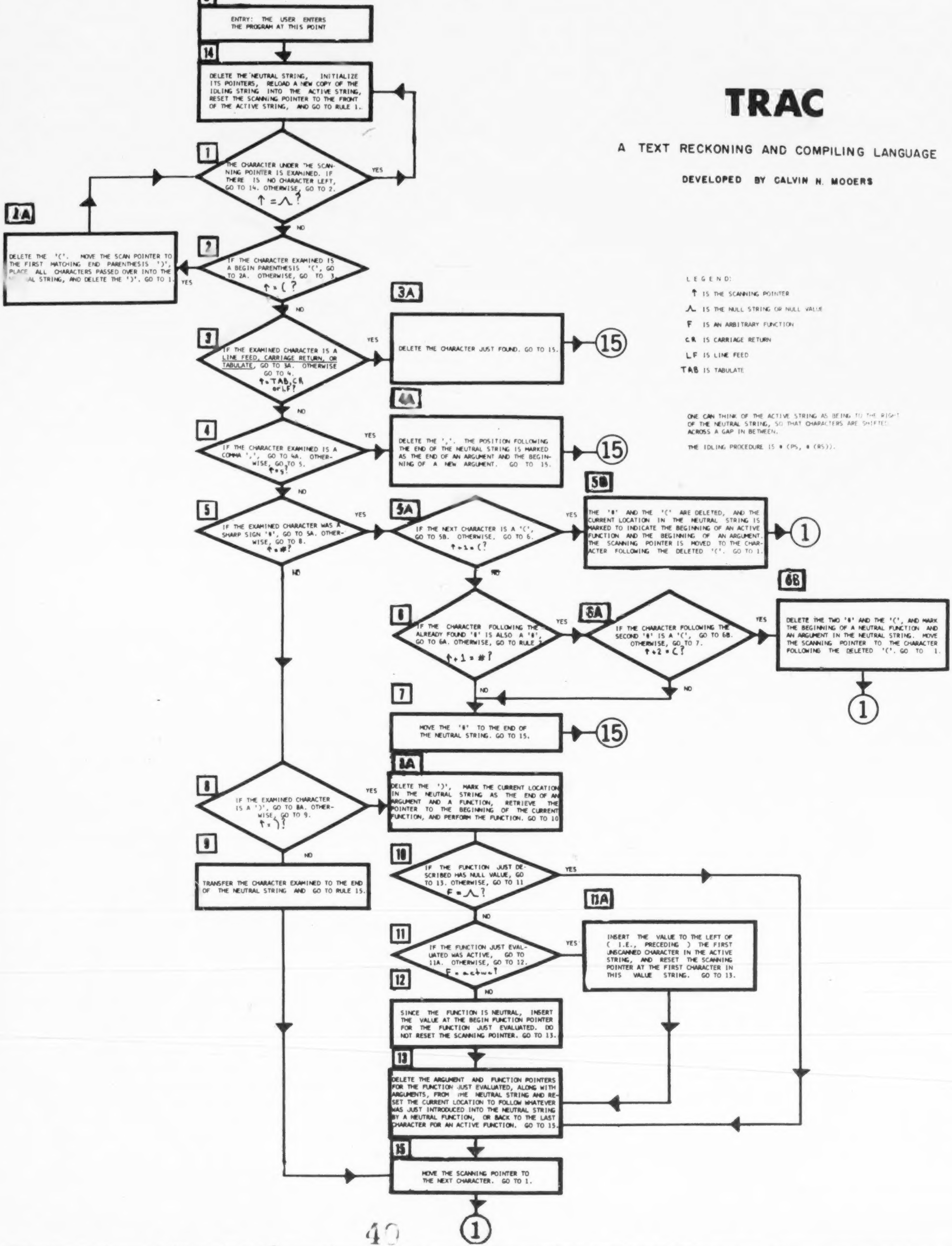
DEVELOPED BY CALVIN N. MOOERS

LEGEND:

- ↑ IS THE SCANNING POINTER
- ʌ IS THE NULL STRING OR NULL VALUE
- F IS AN ARBITRARY FUNCTION
- CR IS CARRIAGE RETURN
- LF IS LINE FEED
- TAB IS TABULATE

ONE CAN THINK OF THE ACTIVE STRING AS BEING TO THE RIGHT OF THE NEUTRAL STRING, SO THAT CHARACTERS ARE SHIFTED ACROSS A GAP IN BETWEEN.

THE IDLING PROCEDURE IS # (PS, # (RS)).



0 ENTRY: THE USER ENTERS THE PROGRAM AT THIS POINT

14 DELETE THE NEUTRAL STRING, INITIALIZE ITS POINTERS, RELOAD A NEW COPY OF THE IDLING STRING INTO THE ACTIVE STRING, RESET THE SCANNING POINTER TO THE FRONT OF THE ACTIVE STRING, AND GO TO RULE 1.

1 THE CHARACTER UNDER THE SCANNING POINTER IS EXAMINED. IF THERE IS NO CHARACTER LEFT, GO TO 14. OTHERWISE, GO TO 2. ↑ = ʌ ?

2 IF THE CHARACTER EXAMINED IS A BEGIN PARENTHESIS '(', GO TO 2A. OTHERWISE, GO TO 3. ↑ = (?

2A DELETE THE '('. MOVE THE SCAN POINTER TO THE FIRST MATCHING END PARENTHESIS ')', PLACE ALL CHARACTERS PASSED OVER INTO THE NEUTRAL STRING, AND DELETE THE ')'. GO TO 1.

3 IF THE EXAMINED CHARACTER IS A LINE FEED, CARRIAGE RETURN, OR TABULATE, GO TO 3A. OTHERWISE GO TO 4. ↑ = TAB, CR or LF?

3A DELETE THE CHARACTER JUST FOUND. GO TO 15.

4 IF THE CHARACTER EXAMINED IS A COMMA ',', GO TO 4A. OTHERWISE, GO TO 5. ↑ = , ?

4A DELETE THE ','. THE POSITION FOLLOWING THE END OF THE NEUTRAL STRING IS MARKED AS THE END OF AN ARGUMENT AND THE BEGINNING OF A NEW ARGUMENT. GO TO 15.

5 IF THE EXAMINED CHARACTER WAS A SHARP SIGN '#', GO TO 5A. OTHERWISE, GO TO 8. ↑ = # ?

5A IF THE NEXT CHARACTER IS A '(', GO TO 5B. OTHERWISE, GO TO 6. ↑ + 1 = (?

5B THE '#' AND THE '(' ARE DELETED, AND THE CURRENT LOCATION IN THE NEUTRAL STRING IS MARKED TO INDICATE THE BEGINNING OF AN ACTIVE FUNCTION AND THE BEGINNING OF AN ARGUMENT. THE SCANNING POINTER IS MOVED TO THE CHARACTER FOLLOWING THE DELETED '('. GO TO 1.

6 IF THE CHARACTER FOLLOWING THE ALREADY FOUND '#' IS ALSO A '#', GO TO 6A. OTHERWISE, GO TO RULE 7. ↑ + 1 = # ?

6A IF THE CHARACTER FOLLOWING THE SECOND '#' IS A '(', GO TO 6B. OTHERWISE, GO TO 7. ↑ + 2 = (?

6B DELETE THE TWO '#' AND THE '(', AND MARK THE BEGINNING OF A NEUTRAL FUNCTION AND AN ARGUMENT IN THE NEUTRAL STRING. MOVE THE SCANNING POINTER TO THE CHARACTER FOLLOWING THE DELETED '('. GO TO 1.

7 MOVE THE '#' TO THE END OF THE NEUTRAL STRING. GO TO 15.

8 IF THE EXAMINED CHARACTER IS A ')', GO TO 8A. OTHERWISE, GO TO 9. ↑ =) ?

8A DELETE THE ')', MARK THE CURRENT LOCATION IN THE NEUTRAL STRING AS THE END OF AN ARGUMENT AND A FUNCTION, RETRIEVE THE POINTER TO THE BEGINNING OF THE CURRENT FUNCTION, AND PERFORM THE FUNCTION. GO TO 10

9 TRANSFER THE CHARACTER EXAMINED TO THE END OF THE NEUTRAL STRING AND GO TO RULE 15.

10 IF THE FUNCTION JUST DESCRIBED HAS NULL VALUE, GO TO 13. OTHERWISE, GO TO 11 F = ʌ ?

11 IF THE FUNCTION JUST EVALUATED WAS ACTIVE, GO TO 11A. OTHERWISE, GO TO 12. F = active?

11A INSERT THE VALUE TO THE LEFT OF (I.E., PRECEDING) THE FIRST UNSCANNED CHARACTER IN THE ACTIVE STRING, AND RESET THE SCANNING POINTER AT THE FIRST CHARACTER IN THIS VALUE STRING. GO TO 13.

12 SINCE THE FUNCTION IS NEUTRAL, INSERT THE VALUE AT THE BEGIN FUNCTION POINTER FOR THE FUNCTION JUST EVALUATED. DO NOT RESET THE SCANNING POINTER. GO TO 13.

13 DELETE THE ARGUMENT AND FUNCTION POINTERS FOR THE FUNCTION JUST EVALUATED, ALONG WITH ARGUMENTS, FROM THE NEUTRAL STRING AND RESET THE CURRENT LOCATION TO FOLLOW WHATEVER WAS JUST INTRODUCED INTO THE NEUTRAL STRING BY A NEUTRAL FUNCTION, OR BACK TO THE LAST CHARACTER FOR AN ACTIVE FUNCTION. GO TO 15.

15 MOVE THE SCANNING POINTER TO THE NEXT CHARACTER. GO TO 1.

APPENDIX B

ELIZA -- A Computer Program for the Study of Natural Language Communication Between Man and Machine

and

Contextual Understanding by Computers

Joseph Weizenbaum
Massachusetts Institute of Technology

included with permission of the author

Computational Linguistics

A. G. OETTINGER, Editor

ELIZA—A Computer Program For the Study of Natural Language Communication Between Man And Machine

JOSEPH WEIZENBAUM
Massachusetts Institute of Technology, Cambridge, Mass.*

ELIZA is a program operating within the MAC time-sharing system at MIT which makes certain kinds of natural language conversation between man and computer possible. Input sentences are analyzed on the basis of decomposition rules which are triggered by key words appearing in the input text. Responses are generated by reassembly rules associated with selected decomposition rules. The fundamental technical problems with which ELIZA is concerned are: (1) the identification of key words, (2) the discovery of minimal context, (3) the choice of appropriate transformations, (4) generation of responses in the absence of key words, and (5) the provision of an editing capability for ELIZA "scripts". A discussion of some psychological issues relevant to the ELIZA approach as well as of future developments concludes the paper.

Introduction

It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the area of computer programming, especially in what is called heuristic programming and artificial intelligence. For in those realms machines are made to behave in wondrous ways, often sufficient to dazzle even the most experienced observer. But once a particular program is unmasked, once its inner workings are explained in language sufficiently plain to induce understanding, its magic crumbles away; it stands revealed as a mere collection of procedures, each quite comprehensible. The observer says to himself "I could have written that". With that thought he moves the program in question from the shelf marked "intelligent", to that reserved for curios, fit to be discussed only with people less enlightened than he.

The object of this paper is to cause just such a re-evaluation of the program about to be "explained". Few programs ever needed it more.

ELIZA Program

ELIZA is a program which makes natural language conversation with a computer possible. Its present implementation is on the MAC time-sharing system at MIT. It is written in MAD-SLIP [4] for the IBM 7094. Its name was chosen to emphasize that it may be incrementally improved by its users, since its language abilities may be continually improved by a "teacher". Like the Eliza of Pygmalion fame, it can be made to appear even more civilized, the relation of appearance to reality, however, remaining in the domain of the playwright.

For the present purpose it is sufficient to characterize the MAC system as one which permits an individual to operate a full scale computer from a remotely located typewriter. The individual operator has the illusion that he is the sole user of the computer complex, while in fact others may be "time-sharing" the system with him. What is important here is that the computer can read messages typed on the typewriter and respond by writing on the same instrument. The time between the computer's receipt of a message and the appearance of its response is a function of the program controlling the dialogue and of such MAC system parameters as the number of users currently corresponding with the system. These latter parameters generally contribute so little to the overall response time that conversational interaction with the computer need never involve truly intolerable delays.

When in conversation with ELIZA, the user types in some statement or set of statements in natural language using normal punctuation and sentence structures. Only the question mark may not be used, because it is interpreted as a line delete character by the MAC system. The user's statement is terminated by a double carriage return which serves to turn control over to ELIZA. ELIZA then analyzes the user's statement and generates some response which it types out. Control is then again with the user.

A typical conversation is the following:

```
Men are all alike.
IN WHAT WAY
They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE
Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
```

Work reported herein was supported (in part) by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

* Department of Electrical Engineering.

It's true. I am unhappy.
DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY
I need some help, that much seems certain.
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
Perhaps I could learn to get along with my mother.
TELL ME MORE ABOUT YOUR FAMILY
My mother takes care of me.
WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
My father.
YOUR FATHER
You are like my father in some ways.
WHAT RESEMBLANCE DO YOU SEE
You are not very aggressive but I think you don't want me to notice that.
WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE
You don't argue with me.
WHY DO YOU THINK I DON'T ARGUE WITH YOU
You are afraid of me.
DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
My father is afraid of everybody.
WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR FATHER
Bullies.
DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR BOYFRIEND MADE YOU COME HERE

The capitalized lines are the machine responses.

The gross procedure of the program is quite simple; the text is read and inspected for the presence of a *keyword*. If such a word is found, the sentence is transformed according to a *rule* associated with the keyword, if not a content-free remark or, under certain conditions, an earlier transformation is retrieved. The text so computed or retrieved is then printed out.

In detail, of course, the procedure sketched above is considerably more complex. Keywords, for example, may have a RANK or precedence number. The procedure is sensitive to such numbers in that it will abandon a keyword already found in the left-to-right scan of the text in favor of one having a higher rank. Also, the procedure recognizes a comma or a period as a delimiter. Whenever either one is encountered and a keyword has already been found, all subsequent text is deleted from the input message. If no key had yet been found the phrase or sentence to the left of the delimiter (as well as the delimiter itself) is deleted. As a result, only single phrases or sentences are ever transformed.

Keywords and their associated transformation[1] rules constitute the SCRIPT for a particular class of conversation. An important property of ELIZA is that a script is data; i.e., it is not part of the program itself. Hence, ELIZA is not restricted to a particular set of recognition patterns or responses, indeed not even to any specific language. ELIZA scripts exist (at this writing) in Welsh and German as well as in English.

The fundamental technical problems with which ELIZA must be preoccupied are the following:

(1) The identification of the "most important" keyword occurring in the input message.

(2) The identification of some minimal context within which the chosen keyword appears; e.g., if the keyword is "you", is it followed by the word "are" (in which case an assertion is probably being made).

(3) The choice of an appropriate transformation rule and, of course, the making of the transformation itself.

(4) The provision of mechanism that will permit ELIZA to respond "intelligently" when the input text contained no keywords.

(5) The provision of machinery that facilitates editing, particularly extension, of the script on the script writing level.

There are, of course, the usual constraints dictated by the need to be economical in the use of computer time and storage space.

The central issue is clearly one of text manipulation, and at the heart of that issue is the concept of the *transformation rule* which has been said to be associated with certain keywords. The mechanisms subsumed under the slogan "transformation rule" are a number of SLIP functions which serve to (1) decompose a data string according to certain criteria, hence to test the string as to whether it satisfies these criteria or not, and (2) to reassemble a decomposed string according to certain assembly specifications.

While this is not the place to discuss these functions in all their detail (or even to reveal their full power and generality), it is important to the understanding of the operation of ELIZA to describe them in *some* detail.

Consider the sentence "I am very unhappy these days". Suppose a foreigner with only a limited knowledge of English but with a very good ear heard that sentence spoken but understood only the first two words "I am". Wishing to appear interested, perhaps even sympathetic, he may reply "How long have you been very unhappy these days?" What he must have done is to apply a kind of template to the original sentence, one part of which matched the two words "I am" and the remainder isolated the words "very unhappy these days". He must also have a reassembly kit specifically associated with that template, one that specifies that any sentence of the form "I am BLAH" can be transformed to "How long have you been BLAH", independently of the meaning of BLAH. A somewhat more complicated example is given by the sentence "It seems that you hate me". Here the foreigner understands only the words "you" and "me"; i.e., he applies a template that decomposes the sentence into the four parts:

(1) It seems that (2) you (3) hate (4) me

of which only the second and fourth parts are understood. The reassembly rule might then be "What makes you think I hate you"; i.e., it might throw away the first component, translate the two known words ("you" to "I" and "me" to "you") and tack on a stock phrase (What makes you think) to the front of the reconstruction.

---

[1] The word "transformation" is used in its generic sense rather than that given it by Harris and Chomsky in linguistic contexts.

A formal notation in which to represent the decomposition template is:

(0 YOU 0 ME)

and the reassembly rule

(WHAT MAKES YOU THINK I 3 YOU).

The "0" in the decomposition rule stands for "an indefinite number of words" (analogous to the indefinite dollar sign of COMIT) [6] while the "3" in the reassembly rule indicates that the third component of the subject decomposition is to be inserted in its place. The decomposition rule

(0 YOU 1 ME)

would have worked just as well in this specific example. A nonzero integer "n" appearing in a decomposition rule indicates that the component in question should consist of exactly "n" words. However, of the two rules shown, only the first would have *matched* the sentence, "It seems you hate and love me," the second failing because there is more than one word between "you" and "me".

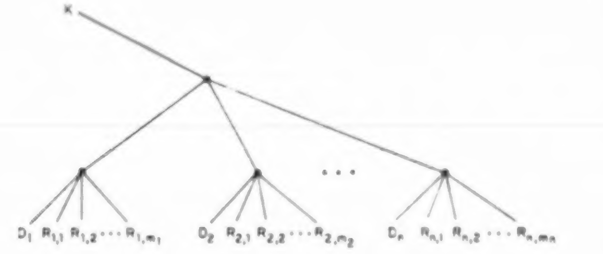

K
D_1 $R_{1,1}$ $R_{1,2}$ ⋯ R_{1,m_1}
D_2 $R_{2,1}$ $R_{2,2}$ ⋯ R_{2,m_2}
D_n $R_{n,1}$ $R_{n,2}$ ⋯ R_{n,m_n}

FIG. 1. Keyword and rule list structure

In ELIZA the question of which decomposition rules to apply to an input text is of course a crucial one. The input sentence might have been, for example, "It seems that you hate," in which case the decomposition rule (0 YOU 0 ME) would have failed in that the word "ME" would not have been found at all, let alone in its assigned place. Some other decomposition rule would then have to be tried and, failing that, still another until a match could be made or a total failure reported. ELIZA must therefore have a mechanism to sharply delimit the set of decomposition rules which are potentially applicable to a currently active input sentence. This is the keyword mechanism.

An input sentence is scanned from left to right. Each word is looked up in a dictionary of keywords. If a word is identified as a keyword, then (apart from the issue of precedence of keywords) only decomposition rules containing that keyword need to be tried. The trial sequence can even be partially ordered. For example, the decomposition rule (0 YOU 0) associated with the keyword "YOU" (and decomposing an input sentence into (1) all words in front of "YOU", (2) the word "YOU", and (3) all words following "YOU") should be the last one tried since it is bound to succeed.

Two problems now arise. One stems from the fact that almost none of the words in any given sentence are represented in the keyword dictionary. The other is that of "associating" both decomposition and reassembly rules with keywords. The first is serious in that the determination that a word is not in a dictionary may well require more computation (i.e., time) than the location of a word which is represented. The attack on both problems begins by placing both a keyword and its associated rules on a *list*. The basic format of a typical key list is the following:

$$\begin{array}{ll} (K & ((D_1)\ (R_{1,1})\ (R_{1,2})\ \cdots\ (R_{1,m_1})) \\ & ((D_2)\ (R_{2,1})\ (R_{2,2})\ \cdots\ (R_{2,m_2})) \\ & \quad \vdots \qquad\qquad\qquad\qquad\qquad \vdots \\ & ((D_n)\ (R_{n,1})\ (R_{n,2})\ \cdots\ (R_{n,m_n}))) \end{array}$$

where K is the keyword, D_i the ith decomposition rule associated with K and $R_{i,j}$ the jth reassembly rule associated with the ith decomposition rule.

A common pictorial representation of such a structure is the tree diagram shown in Figure 1. The top level of this structure contains the keyword followed by the names of lists; each one of which is again a list structure beginning with a decomposition rule and followed by reassembly rules. Since list structures of this type have no predetermined dimensionality limitations, any number of decomposition rules may be associated with a given keyword and any number of reassembly rules with any specific decomposition rule. SLIP is rich in functions that sequence over structures of this type efficiently. Hence programming problems are minimized.

An ELIZA script consists mainly of a set of list structures of the type shown. The actual keyword dictionary is constructed when such a script is first read into the hitherto empty program. The basic structural component of the keyword dictionary is a vector KEY of (currently) 128 contiguous computer words. As a particular key list structure is read the keyword K at its top is randomized (hashed) by a procedure that produces (currently) a 7 bit integer "i". The word "always", for example, yields the integer 14. KEY(i), i.e., the ith word of the vector KEY, is then examined to determine whether it contains a list name. If it does not, then an empty list is created, its name placed in KEY(i), and the key list structure in question placed on *that* list. If KEY(i) already contains a list name, then the name of the key list structure is placed on the bottom of the list named in KEY(i). The largest dictionary so far attempted contains about 50 keywords. No list named in any of the words of the KEY vector contains more than two key list structures.

Every word encountered in the scan of an input text, i.e., during the actual operations of ELIZA, is randomized by the same hashing algorithm as was originally applied to the incoming keywords, hence yields an integer which points to the only possible list structure which could potentially contain that word as a keyword. Even then, only the tops of any key list structures that may be found there need be interrogated to determine whether or not a keyword has been found. By virtue of the various list sequencing operations that SLIP makes available, the actual identification of a keyword leaves as its principal product a pointer to the list of decomposition (and hence reassembly) rules associated with the identified keyword. One result of this strategy is that often less time is required to discover that a given word is not in the keyword dictionary than to locate it if it is there. However, the location of a keyword yields pointers to all information associated with that word.

Some conversational protocols require that certain transformations be made on certain words of the input text independently of any contextual considerations. The first conversation displayed in this paper, for example, requires that first person pronouns be exchanged for second person pronouns and vice versa throughout the input text. There may be further transformations but these minimal substitutions are unconditional. Simple substitution rules ought not to be elevated to the level of transformations, nor should the words involved be forced to carry with them all the structure required for the fully complex case. Furthermore, unconditional substitutions of single words for single words can be accomplished during the text scan itself, not as a transformation of the entire text subsequent to scanning. To facilitate the realization of these desiderata, any word in the key dictionary, i.e., at the top of a key list structure, may be followed by an equal sign followed by whatever word is to be its substitute. Tranformation rules may, but need not, follow. If none do follow such a substitution rule, then the substitution is made on the fly, i.e., during text scanning, but the word in question is not identified as a keyword for subsequent purposes. Of course, a word may be both subtituted for and be a keyword as well. An example of a simple substitution is

(YOURSELF = MYSELF).

Neither "yourself" nor "myself" are keywords in the particular script from which this example was chosen.

The fact that keywords can have ranks or precedences has already been mentioned. The need of a ranking mechanism may be established by an example. Suppose an input sentence is "I know everybody laughed at me." A script may tag the word "I" as well as the word "everybody" as a keyword. Without differential ranking, "I" occurring first would determine the transformation to be applied. A typical response might be "You say you know everybody laughed at you." But the important message in the input sentence begins with the word "everybody". It is very often true that when a person speaks in terms of universals such as "everybody", "always" and "nobody" he is really referring to some quite specific event or person. By giving "everybody" a higher rank than "I", the response "Who in particular are you thinking of" may be generated.

The specific mechanism employed in ranking is that the rank of every keyword encountered (absence of rank implies rank equals 0) is compared with the rank of the highest ranked keyword already seen. If the rank of the new word is higher than that of any previously encountered word, the pointer to the transformation rules associated with the new word is placed on top of a list called the keystack, otherwise it is placed on the bottom of the keystack. When the text scan terminates, the keystack has at its top a pointer associated with the highest ranked keyword encountered in the scan. The remaining pointers in the stack may not be monotonically ordered with respect to the ranks of the words from which they were derived, but they are nearly so—in any event they are in a useful and interesting order. Figure 2 is a simplified flow diagram of keyword detection. The rank of a keyword must, of course, also be associated with the keyword. Therefore it must appear on the keyword list structure. It may be found, if at all, just in front of the list of transformation rules associated with the keyword. As an example consider the word "MY" in a particular script. Its keyword list may be as follows:

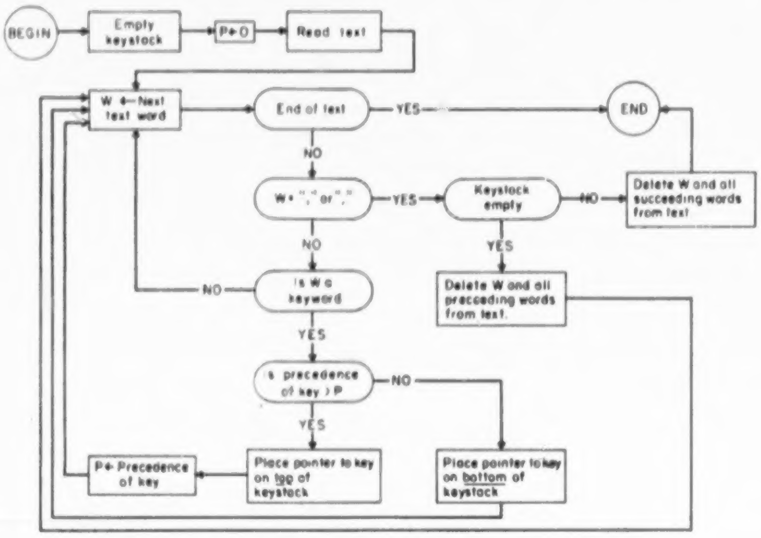


FIG. 2. Basic flow diagram of keyword detection

(MY = YOUR 5 (transformation rules)).

Such a list would mean that whenever the word "MY" is encountered in any text, it would be replaced by the word "YOUR". Its rank would be 5.

Upon completion of a given text scan, the keystack is either empty or contains pointers derived from the keywords found in the text. Each of such pointers is actually a sequence reader—a SLIP mechanism which facilitates scanning of lists—pointing into its particular key list in such a way that one sequencing operation to the right (SEQLR) will sequence it to the first set of transformation rules associated with its keyword, i.e., to the list

$$((D_1)\ (R_{1,1})\ (R_{1,2})\ \ldots\ (R_1\ ,\ R_{m_1})).$$

The top of *that* list, of course, is a list which serves a decomposition rule for the subject text. The top of the keystack contains the first pointer to be activated.

The decomposition rule D_1 associated with the keyword K, i.e., $\{(D_1), K\}$, is now tried. It may fail however. For example, suppose the input text was:

You are very helpful.

The keyword, say, is "you", and $\{(D_1), \text{you}\}$ is

(0 I remind you of 0).

(Recall that the "you" in the original sentence has already been replaced by "I" in the text now analyzed.) This decomposition rule obviously fails to match the input sentence. Should $\{(D_1), K\}$ fail to find a match, then $\{(D_2), K\}$ is tried. Should that too fail, $\{(D_3), K\}$ is attempted, and so on. Of course, the set of transformation rules can be guaranteed to terminate with a decomposition rule which must match. The decomposition rule

(0 K 0)

will match any text in which the word K appears while

(0)

will match any text whatever. However, there are other ways to leave a particular set of transformation rules, as will be shown below. For the present, suppose that some particular decomposition rule (D_i) has matched the input text. (D_i), of course, was found on a list of the form

$$((D_i)(R_{i,1})(R_{i,2}) \cdots (R_{i,m_i})).$$

Sequencing the reader which is presently pointing at (D_i) will retrieve the reassembly rule $(R_{i,1})$ which may then be applied to the decomposed input text to yield the output message.

Consider again the input text

You are very helpful

in which "you" is the only key word. The sentence is transformed during scanning to

I are very helpful

$\{(D_1), \text{you}\}$ is "(0 I remind your of 0)" and fails to match as already discussed. However, $\{(D_2), \text{you}\}$ is "(0 I are 0)" and obviously matches the text, decomposing it into the constituents

(1) empty (2) I (3) are (4) very helpful.

$\{(R_{2,1}), \text{you}\}$ is

(What makes you think I am 4)

Hence it produces the output text

What makes you think I am very helpful.

Having produced it, the integer 1 is put in front of $(R_{2,1})$ so that the transformation rule list in question now appears as

$$((D_2)1(R_{2,1})(R_{2,2}) \cdots (R_{2,m_2})).$$

Next time $\{(D_2), K\}$ matches an input text, the reassembly rule $(R_{2,2})$ will be applied and the integer 2 will replace the 1. After (R_{2,m_2}) has been exercised, $(R_{2,1})$ will again be invoked. Thus, after the system has been in use for a time, every decomposition rule which has matched some input text has associated with it an integer which corresponds to the last reassembly rule used in connection with that decomposition rule. This mechanism insures that the complete set of reassembly rules associated with a given decomposition rule is cycled through before any repetitions occur.

The system described so far is essentially one which selects a decomposition rule for the highest ranking keyword found in an input text, attempts to match that text according to that decomposition rule and, failing to make a match, selects the next reassembly rule associated with the matching decomposition rule and applies it to generate an output text. It is, in other words, a system which, for the highest ranking keyword of a text, selects a specific decomposition and reassembly rule to be used in forming the output message.

Were the system to remain that simple, then keywords that required identical sets of transformation rules would each require that a copy of these transformation rules be associated with them. This would be logically sound but would complicate the task of script writing and would also make unnecessary storage demands. There are therefore special types of decomposition and assembly rules characterized by the appearance of "=" at the top of the rule list. The word following the equal sign indicates which new set of transformation rules is to be applied. For example, the keyword "what" may have associated with it a transformation rule set of the form

((0) (Why do you ask) (Is that an important question) . . .)

which would apply equally well to the keywords "how" and "when". The entire keyword list for "how" may therefore be

(How (=What))

The keywords "how", "what" and "when" may thus be made to form an equivalence class with respect to the transformation rules which are to apply to them.

In the above example the rule "(=what)" is in the place of a decomposition rule, although it causes no decomposition of the relevant text. It may also appear, however, in the place of a reassembly rule. For example, the keyword "am" may have among others the following transformation rule set associated with it:

((0 are you 0) (Do you believe you are 4) . . . (=what) . . .)

(It is here assumed that "are" has been substituted for "am" and "you" for "I" in the initial text scan.) Then, the input text

Am I sick

would elicit either

Do you believe you are sick

or

Why do you ask

depending on how many times the general form had already occurred.

Under still other conditions it may be desirable to perform a preliminary transformation on the input text before subjecting it to the decompositions and reassemblies which finally yield the output text. For example, the keyword "you're" should lead to the transformation rules associated with "you" but should first be replaced by a word pair. The dictionary entry for "you're" is therefore:

```
(you're = I'm ((0 I'm 0) (PRE (I AM 3) (=YOU))))
```

which has the following effect:

(1) Wherever "you're" is found in the input text, it is replaced by "I'm".

(2) If "you're" is actually selected as the regnant keyword, then the input text is decomposed into three constituent parts, namely, all text in front of the first occurrence of "I'm", the word "I'm" itself, and all text following the first occurrence of "I'm".

(3) The reassembly rule beginning with the code "PRE" is encountered and the decomposed text reassembled such that the words 'I AM" appear in front of the third constituent determined by the earlier decomposition.

(4) Control is transferred, so to speak, to the transformation rules associated with the keyword "you", where further decompositions etc. are attempted.

It is to be noted that the set

```
(PRE (I AM 3) (=YOU))
```

is logically in the place of a reassembly rule and may therefore be one of many reassembly rules associated with the given decomposition.

Another form of reassembly rule is

```
(NEWKEY)
```

which serves the case in which attempts to match on the currently regnant keyword are to be given up and the entire decomposition and reassembly process is to start again on the basis of the keyword to be found in the keystack. Whenever this rule is invoked, the top of the keystack is "popped up" once, i.e., the new regnant keyword recovered and removed from the keystack, and the entire process reinitiated as if the initial text scan had just terminated. This mechanism makes it possible to, in effect, test on key phrases as opposed to single key words.

A serious problem which remains to be discussed is the reaction of the system in case no keywords remain to serve as transformation triggers. This can arise either in case the keystack is empty when NEWKEY is invoked or when the input text contained no keywords initially.

The simplest mechanism supplied is in the form of the special reserved keyword "NONE" which must be part of any script. The script writer must associate the universally matching decomposition rule (0) with it and follow this by as many content-free remarks in the form of transformation rules as he pleases. (Examples are: "Please go on", "That's very interesting" and "I see".)

There is, however, another mechanism which causes the system to respond more spectacularly in the absence of a key. The word "MEMORY" is another reserved pseudo-keyword. The key list structure associated with it differs from the ordinary one in some respects. An example illuminates this point.

Consider the following structure:

```
(MEMORY MY
  (0 YOUR 0 = LETS DISCUSS FURTHER WHY YOUR 3)
  (0 YOUR 0 = EARLIER YOU SAID YOUR 3)
  ⋮
```

The word "MY" (which must be an ordinary keyword as well) has been selected to serve a special function. Whenever it is the highest ranking keyword of a text one of the transformations on the MEMORY list is randomly selected, and a copy of the text is transformed accordingly. This transformation is stored on a first-in-first-out stack for later use. The ordinary processes already described are then carried out. When a text without keywords is encountered later and a certain counting mechanism is in a particular state and the stack in question is not empty, then the transformed text is printed out as the reply. It is, of course, also deleted from the stack of such transformations.

The current version of ELIZA requires that one keyword be associated with MEMORY and that exactly four transformations accompany that word in that context. (An application of a transformation rule of the form

```
(LEFT HAND SIDE = RIGHT HAND SIDE)
```

is equivalent to the successive application of the two forms

```
(LEFT HAND SIDE), (RIGHT HAND SIDE).)
```

Three more details will complete the formal description of the ELIZA program.

The transformation rule mechanism of SLIP is such that it permits tagging of words in a text and their subsequent recovery on the basis of one of their tags. The keyword "MOTHER" in ELIZA, for example, may be identified as a noun and as a member of the class "family" as follows:

```
(MOTHER DLIST (/NOUN FAMILY)).
```

Such tagging in no way interferes with other information (e.g., rank or transformation rules) which may be associated with the given tag word. A decomposition rule may contain a matching constituent of the form (/TAG1 TAG2 ···) which will match and isolate a word in the subject text having any one of the mentioned tags. If, for example, "MOTHER" is tagged as indicated and the input text

"CONSIDER MY AGED MOTHER AS WELL AS ME"

subjected to the decomposition rule

```
(0 YOUR 0 (/FAMILY) 0)
```

(remembering that "MY" has been replaced by "YOUR"), then the decomposition would be

(1) CONSIDER (2) YOUR (3) AGED (4) MOTHER
(5) AS WELL AS ME.

Another flexibility inherent in the SLIP text manipulation mechanism underlying ELIZA is tha or-ing of matching criteria is permitted in decomposition rules. The above input text would have been decomposed precisely as stated above by the decomposition rule:

(0 YOUR 0 (*FATHER MOTHER) 0)

which, by virtue of the presence of "*" in the sublist structure seen above, would have isolated either the word "FATHER" or "MOTHER" (in that order) in the input text, whichever occurred first after the first appearance of the word "YOUR".

Finally, the script writer must begin his script with a list, i.e., a message enclosed in parentheses, which contains the statement he wishes ELIZA to type when the system is first loaded. This list may be empty.

Editing of an ELIZA script is achieved via appeal to a contextual editing program (ED) which is part of the MAC library. This program is called whenever the input text to ELIZA consists of the single word "EDIT". ELIZA then puts itself in a so-called dormant state and presents the then stored script for editing. Detailed description of ED is out of place here. Suffice it to say that changes, additions and deletions of the script may be made with considerable efficiency and on the basis of entirely contextual cues, i.e., without resort to line numbers or any other artificial devices. When editing is completed, ED is given the command to FILE the revised script. The new script is then stored on the disk and read into ELIZA. ELIZA then types the word "START" to signal that the conversation may resume under control of the new script.

An important consequence of the editing facility built into ELIZA is that a given ELIZA script need not start out to be a large, full-blown scenario. On the contrary, it should begin as a quite modest set of keywords and transformation rules and permitted to be grown and molded as experience with it builds up. This appears to be the best way to use a truly interactive man-machine facility—i.e., not as a device for rapidly debugging a code representing a fully thought out solution to a problem, but rather as an aid for the exploration of problem solving strategies.

Discussion

At this writing, the only serious ELIZA scripts which exist are some which cause ELIZA to respond roughly as would certain psychotherapists (Rogerians). ELIZA performs best when its human correspondent is initially instructed to "talk" to it, via the typewriter of course, just as one would to a psychiatrist. This mode of conversation was chosen because the psychiatric interview is one of the few examples of categorized dyadic natural language communication in which one of the participating pair is free to assume the pose of knowing almost nothing of the real world. If, for example, one were to tell a psychiatrist "I went for a long boat ride" and he responded "Tell me about boats", one would not assume that he knew nothing about boats, but that he had some purpose in so directing the subsequent conversation. It is important to note that this assumption is one made by the speaker. Whether it is realistic or not is an altogether separate question. In any case, it has a crucial psychological utility in that it serves the speaker to maintain his sense of being heard and understood. The speaker further defends his impression (which even in real life may be illusory) by attributing to his conversational partner all sorts of background knowledge, insights and reasoning ability. But again, these are the *speaker's* contribution to the conversation. They manifest themselves inferentially in the *interpretations* he makes of the offered responses. From the purely technical programming point of view then, the psychiatric interview form of an ELIZA script has the advantage that it eliminates the need of storing *explicit* information about the real world.

The human speaker will, as has been said, contribute much to clothe ELIZA'S responses in vestments of plausibility. But he will not defend his illusion (that he is being understood) against all odds. In human conversation a speaker will make certain (perhaps generous) assumptions about his conversational partner. As long as it remains possible to interpret the latter's responses consistently with those assumptions, the speaker's image of his partner remains unchanged, in particular, undamaged. Responses which are difficult to so interpret may well result in an enhancement of the image of the partner, in additional rationalizations which then make more complicated interpretations of his responses reasonable. When, however, such rationalizations become too massive and even self-contradictory, the entire image may crumble and be replaced by another ("He is not, after all, as smart as I thought he was"). When the conversational partner is a machine (the distinction between machine and program is here not useful) then the idea of *credibility* may well be substituted for that of *plausibility* in the above.

With ELIZA as the basic vehicle, experiments may be set up in which the subjects find it credible to believe that the responses which appear on his typewriter are generated by a human sitting at a similar instrument in another room. How must the script be written in order to mantain the credibility of this idea over a long period of time? How can the performance of ELIZA be systematically degraded in order to achieve controlled and predictable thresholds of credibility in the subject? What, in all this, is the role of the initial instruction to the subject? On the other hand, suppose the subject is told he is communicating with a machine. What is he led to believe about the machine as a result of his conversational experience with it? Some subjects have been very hard to convince that ELIZA (with its present script) is *not* human. This is a striking form of Turing's test. What experimental design would make it more nearly rigorous and airtight?

The whole issue of the credibility (to humans) of machine output demands investigation. Important decisions increasingly tend to be made in response to computer output. The ultimately responsible human interpreter of "What the machine says" is, not unlike the correspondent with ELIZA, constantly faced with the need to make credibility judgments. ELIZA shows, if nothing else, how easy it is to create and maintain the illusion of understanding, hence perhaps of judgment deserving of credibility. A certain danger lurks there.

The idea that the present ELIZA script contains *no* information about the real world is not entirely true. For example, the transformation rules which cause the input

Everybody hates me

to be transformed to

Can you think of anyone in particular

and other such are based on quite specific hypotheses about the world. The whole script constitutes, in a loose way, a model of certain aspects of the world. The act of writing a script is a kind of programming act and has all the advantages of programming, most particularly that it clearly shows where the programmer's understanding and command of his subject leaves off.

A large part of whatever elegance may be credited to ELIZA lies in the fact that ELIZA maintains the illusion of understanding with so little machinery. But there are bounds on the extendability of ELIZA's "understanding" power, which are a function of the ELIZA program itself and not a function of any script it may be given. The crucial test of understanding, as every teacher should know, is not the subject's ability to continue a conversation, but to draw valid conclusions from what he is being told. In order for a computer program to be able to do that, it must at least have the capacity to store *selected* parts of its inputs. ELIZA throws away each of its inputs, except for those few transformed by means of the MEMORY machinery. Of course, the problem is more than one of storage. A great part of it is, in fact, subsumed under the word "selected" used just above. ELIZA in its use so far has had as one of its principal objectives the *concealment* of its lack of understanding. But to encourage its conversational partner to offer inputs from which it can select remedial information, it must *reveal* its misunderstanding. A switch of objectives from the concealment to the revelation of misunderstanding is seen as a precondition to making an ELIZA-like program the basis for an effective natural language man-machine communication system.

One goal for an augmented ELIZA program is thus a system which already has access to a store of information about some aspects of the real world and which, by means of conversational interaction with people, can reveal both what it knows, i.e., behave as an information retrieval system, and where its knowledge ends and needs to be augmented. Hopefully the augmentation of its knowledge will also be a direct consequence of its conversational experience. It is precisely the prospect that such a program will converse with many people and learn something from each of them, which leads to the hope that it will prove an interesting and even useful conversational partner.

One way to state a slightly different intermediate goal is to say that ELIZA should be given the power to slowly build a model of the subject conversing with it. If the subject mentions that he is not married, for example, and later speaks of his wife, then ELIZA should be able to make the tentative inference that he is either a widower or divorced. Of course, he could simply be confused. In the long run, ELIZA should be able to build up a *belief structure* (to use Abelson's phrase) of the subject and on that basis detect the subject's rationalizations, contradictions, etc. Conversations with such an ELIZA would often turn into arguments. Important steps in the realization of these goals have already been taken. Most notable among these is Abelson's and Carroll's work on simulation of belief structures [1].

The script that has formed the basis for most of this discussion happens to be one with an overwhelmingly psychological orientation. The reason for this has already been discussed. There is a danger, however, that the example will run away with what it is supposed to illustrate. It is useful to remember that the ELIZA program itself is merely a translating processor in the technical programming sense. Gorn [2] in a paper on language systems says:

> Given a language which already possesses semantic content, then a translating processor, even if it operates only syntactically, generates corresponding expressions of another language to which we can attribute as "meanings" (possibly multiple—the translator may not be one to one) the "semantic intents" of the generating source expressions; whether we find the result consistent or useful or both is, of course, another problem. It is quite possible that by this method the same syntactic object language can be usefully assigned multiple meanings for each expression . . .

It is striking to note how well his words fit ELIZA. The "given language" is English as is the "other language", expressions of which are generated. In principle, the given language could as well be the kind of English in which "word problems" in algebra are given to high school students and the other language, a machine code allowing a particular computer to "solve" the stated problems. (See Bobrow's program STUDENT [3].)

The intent of the above remarks is to further rob ELIZA of the aura of magic to which its application to psychological subject matter has to some extent contributed. Seen in the coldest possible light, ELIZA is a translating processor in Gorn's sense; however, it is one which has been especially constructed to work well with natural language text.

REFERENCES

1. Abelson, R. P., and Carroll, J. D. Computer simulation of individual belief systems. *Amer. Behav. Sci. 9* (May 1965), 24–30.
2. Gorn, S. Semiotic relationships in ambiguously stratified language systems. Paper presented at Int. Colloq. Algebraic Linguistics and Automatic Theory, Hebrew U. of Jerusalem, Aug. 1964.
3. Bobrow, D. G. Natural language input for a computer problem solving system. Doctoral thesis, Math. Dept., MIT. Cambridge, Mass., 1964.
4. Weizenbaum, J. Symmetric list processor. *Comm. ACM 6,* (Sept. 1963), 524–544.
5. Rogers, C. *Client Centered Therapy: Current Practice, Implications and Theory.* Houghton Mifflin, Boston, 1951.
6. Yngve, J. *COMIT Programming Manual.* MIT Press, Cambridge, Mass., 1961.

APPENDIX. An ELIZA Script

```
(HOW DO YOU DO.  PLEASE TELL ME YOUR PROBLEM)
START
(SORRY ((0) (PLEASE DON'T APOLIGIZE)
(APOLOGIES ARE NOT NECESSARY) (WHAT FEELINGS
DO YOU HAVE WHEN YOU APOLOGIZE) (I'VE TOLD YOU
THAT APOLOGIES ARE NOT REQUIRED)))
(DONT = DON'T)
(CANT = CAN'T)
(WONT = WON'T)
(REMEMBER 5
((0 YOU REMEMBER 0) (DO YOU OFTEN THINK OF 4)
(DOES THINKING OF 4 BRING ANYTHING ELSE TO MIND)
(WHAT ELSE DO YOU REMEMBER)
(WHY DO YOU REMEMBER 4 JUST NOW)
(WHAT IN THE PRESENT SITUATION REMINDS YOU OF 4)
(WHAT IS THE CONNECTION BETWEEN ME AND 4))
((0 DO I REMEMBER 0) (DID YOU THINK I WOULD FORGET 5)
(WHY DO YOU THINK I SHOULD RECALL 5 NOW)
(WHAT ABOUT 5) (=WHAT) (YOU MENTIONED 5))
((0) (NEWKEY)))
(IF 3 ((0 IF 0) (DO YOU THINK ITS LIKELY THAT 3) (DO YOU WISH THAT 3)
(WHAT DO YOU THINK ABOUT 3) (REALLY, 2 3)))
(WHAT DO YOU THINK ABOUT 3) (REALLY, 2 3)))
(DREAMT 4 ((0 YOU DREAMT 0)
(REALLY, 4) (HAVE YOU EVER FANTASIED 4 WHILE YOU WERE AWAKE)
(HAVE YOU DREAMT 4 BEFORE) (=DREAM) (NEWKEY)))
(DREAMED = DREAMT 4 (=DREAMT))
(DREAM 3 ((0) (WHAT DOES THAT DREAM SUGGEST TO YOU)
(DO YOU DREAM OFTEN) (WHAT PERSONS APPEAR IN YOUR DREAMS)
(DON'T YOU BELIEVE THAT DREAM HAS SOMETHING TO DO WITH
YOUR PROBLEM) (NEWKEY)))
(DREAMS = DREAM 3 (=DREAM))
(HOW (=WHAT))
(WHEN (=WHAT))
(ALIKE 10 (=DIT))
(SAME 10 (=DIT))
(CERTAINLY (=YES))
(FEEL DLIST(/BELIEF))
(THINK DLIST(/BELIEF))
(BELIEVE DLIST(/BELIEF))
(WISH DLIST(/BELIEF))
(MEMORY MY (0 YOUR 0 = LETS DISCUSS FURTHER WHY YOUR 3)
(0 YOUR 0 = EARLIER YOU SAID YOUR 3)
(0 YOUR 0 = BUT YOUR 3)
(0 YOUR 0 = DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR 3))
(NONE ((0) (I AM NOT SURE I UNDERSTAND YOU FULLY)
(PLEASE GO ON)
(WHAT DOES THAT SUGGEST TO YOU)
(DO YOU FEEL STRONGLY ABOUT DISCUSSING SUCH THINGS)))
(PERHAPS ((0) (YOU DON'T SEEM QUITE CERTAIN)
(WHY THE UNCERTAIN TONE)
(CAN'T YOU BE MORE POSITIVE)
(YOU AREN'T SURE) (DON'T YOU KNOW)))
(MAYBE (=PERHAPS))
(NAME 15 ((0) (I AM NOT INTERESTED IN NAMES)
(I'VE TOLD YOU BEFORE, I DON'T CARE ABOUT NAMES -
PLEASE CONTINUE)) )
PLEASE CONTINUE)) )
(DEUTSCH (=XFREMD))
(FRANCAIS (=XFREMD))
(ITALIANO (=XFREMD))
(ESPANOL (=XFREMD))
(XFREMD ((0) (I AM SORRY, I SPEAK ONLY ENGLISH)))
(HELLO ((0) (HOW DO YOU DO.  PLEASE STATE YOUR PROBLEM)))
(COMPUTER 50 ((0) (DO COMPUTERS WORRY YOU)
(WHY DO YOU MENTION COMPUTERS) (WHAT DO YOU THINK MACHINES
HAVE TO DO WITH YOUR PROBLEM) (DON'T YOU THINK COMPUTERS CAN
HELP PEOPLE) (WHAT ABOUT MACHINES WORRIES YOU) (WHAT
DO YOU THINK ABOUT MACHINES)))
(MACHINE 50 (=COMPUTER))
(MACHINES 50 (=COMPUTER))
(COMPUTERS 50 (=COMPUTER))
(AM = ARE ((0 ARE YOU 0) (DO YOU BELIEVE YOU ARE 4)
(WOULD YOU WANT TO BE 4) (YOU WISH I WOULD TELL YOU YOU ARE 4)
(WHAT WOULD IT MEAN IF YOU WERE 4) (=WHAT))
((0) (WHY DO YOU SAY 'AM') (I DON'T UNDERSTAND THAT)))
(ARE ((0 ARE I 0 )
(WHY ARE YOU INTERESTED IN WHETHER I AM 4 OR NOT)
(WOULD YOU PREFER IF I WEREN'T 4) (PERHAPS I AM 4 IN YOUR
FANTASIES) (DO YOU SOMETIMES THINK I AM 4) (=WHAT))
((0 ARE 0) (DID YOU THINK THEY MIGHT NOT BE 3)
(WOULD YOU LIKE IT IF THEY WERE NOT 3) (WHAT IF THEY WERE NOT 3)
(POSSIBLY THEY ARE 3)) )
(YOUR = MY ((0 MY 0) (WHY ARE YOU CONCERNED OVER MY 3)
(WHAT ABOUT YOUR OWN 3) (ARE YOU WORRIED ABOUT SOMEONE ELSES 3)
(REALLY, MY 3)))
(WAS 2 ((0 WAS YOU 0 )
(WHAT IF YOU WERE 4) (DO YOU THINK YOU WERE 4)
(WERE YOU 4) (WHAT WOULD IT MEAN IF YOU WERE 4)
(WHAT DOES ' 4 ' SUGGEST TO YOU) (=WHAT))
((0 YOU WAS 0)
(WERE YOU REALLY) (WHY DO YOU TELL ME YOU WERE 4 NOW)
(WERE YOU REALLY) (WHY DO YOU TELL ME YOU WERE 4 NOW)
(PERHAPS I ALREADY KNEW YOU WERE 4) )
((0 WAS I 0) (WOULD YOU LIKE TO BELIEVE I WAS 4)
(WHAT SUGGESTS THAT I WAS 4)
(WHAT DO YOU THINK) (PERHAPS I WAS 4)
(WHAT IF I HAD BEEN 4))
((0) (NEWKEY)) )
(WERE = WAS (=WAS))
(ME = YOU)
(YOU'RE = I'M ((0 I'M 0) (PRE (I ARE 3) (=YOU))))
(I'M = YOU'RE ((0 YOU'RE 0) (PRE (YOU ARE 3) (=I))))
(MYSELF = YOURSELF)
(YOURSELF = MYSELF)
(MOTHER DLIST(/NOUN FAMILY))
(MOM = MOTHER DLIST(/ FAMILY))
(DAD = FATHER DLIST(/ FAMILY))
(FATHER DLIST(/NOUN FAMILY))
(SISTER DLIST(/FAMILY))
(BROTHER DLIST(/FAMILY))
(WIFE DLIST(/FAMILY))
(CHILDREN DLIST(/FAMILY))
(I = YOU
((0 YOU (* WANT NEED) 0) (WHAT WOULD IT MEAN TO YOU IF YOU GOT 4)
(WHY DO YOU WANT 4) (SUPPOSE YOU GOT 4 SOON) (WHAT
IF YOU NEVER GOT 4) (WHAT WOULD GETTING 4 MEAN TO
YOU) (WHAT DOES WANTING 4 HAVE TO DO WITH THIS DISCUSSION))
((0 YOU ARE 0 (*SAD UNHAPPY DEPRESSED SICK ) 0)
(I AM SORRY TO HEAR YOU ARE 5) (DO YOU THINK COMING HERE
WILL HELP YOU NOT TO BE 5) (I'M SURE ITS NOT PLEASANT TO
BE 5) (CAN YOU EXPLAIN WHAT MADE YOU 5))
((0 YOU ARE 0 (*HAPPY ELATED GLAD BETTER ) 0)
(HOW HAVE I HELPED YOU TO BE 5)
(HAS YOUR TREATMENT MADE YOU 5) (WHAT MAKES YOU 5 JUST
NOW) (CAN YOU EXPLAIN WHY YOU ARE SUDDENLY 5))
((0 YOU WAS 0) (=WAS))
((0 YOU WAS 0) (=WAS))
((0 YOU (/BELIEF) YOU 0) (DO YOU REALLY THINK SO) (BUT YOU ARE
NOT SURE YOU 5) (DO YOU REALLY DOUBT YOU 5))
((0 YOU 0 (/BELIEF) 0 I 0) (=YOU))
((0 YOU ARE 0)
(IS IT BECAUSE YOU ARE 4 THAT YOU CAME TO ME)
(HOW LONG HAVE YOU BEEN 4)
(DO YOU BELIEVE IT NORMAL TO BE 4)
(DO YOU ENJOY BEING 4))
((0 YOU (* CAN'T CANNOT) 0) (HOW DO YOU KNOW YOU CAN'T 4)
(HAVE YOU TRIED)
(PERHAPS YOU COULD 4 NOW)
(DO YOU REALLY WANT TO BE ABLE TO 4))
((0 YOU DON'T 0)(DON'T YOU  REALLY 4)(WHY DON'T YOU 4)
(DO YOU WISH TO BE ABLE TO 4) (DOES THAT TROUBLE YOU))
((0 YOU FEEL 0) (TELL ME MORE ABOUT SUCH FEELINGS)
(DO YOU OFTEN FEEL 4)
(DO YOU ENJOY FEELING 4)
(OF WHAT DOES FEELING 4 REMIND YOU))
((0 YOU 0 I 0) (PERHAPS IN YOUR FANTASY WE 3 EACH OTHER)
(DO YOU WISH TO 3 ME)
(YOU SEEM TO NEED TO 3 ME)
(DO YOU 3 ANYONE ELSE))
```

```
((0) (YOU SAY 1)
(CAN YOU ELABORATE ON THAT) (DO YOU SAY 1 FOR SOME SPECIAL REASON)
(THAT'S QUITE INTERESTING)))
(YOU = I ((0 I REMIND YOU OF 0) (=DIT))
((0 I ARE 0) (WHAT MAKES YOU THINK I AM 4)
(DOES IT PLEASE YOU TO BELIEVE I AM 4)
(DO YOU SOMETIMES WISH YOU WERF 4)
(PERHAPS YOU WOULD LIKE TO RE 4))
((0 I 0 YOU) (WHY DO YOU THINK I 3 YOU)
(YOU LIKE TO THINK I 3 YOU - DON'T YOU)
(WHAT MAKES YOU THINK I 3 YOU)
(REALLY, I 3 YOU) (DO YOU WISH TO BFLIEVF I 3 YOU)
(REALLY, I 3 YOU) (DO YOU WISH TO BFLIEVF I 3 YOU)
(SUPPOSE I DID 3 YOU - WHAT WOULD THAT MEAN)
(DOES SOMEONE ELSE BELIEVE I 3 YOU))
((0 I 0) (WE WERE DISCUSSING YOU - NOT ME)
(OH, I 3) (YOU'RE NOT REALLY TALKING ABOUT MF - ARE YOU)
(WHAT ARE YOUR FEELINGS NOW)))
(YES ((0) (YOU SEEM QUITE POSITIVE) (YOU ARE SURE)
(I SEE) (I UNDERSTAND)))
(NO ((0) (ARE YOU SAYING 'NO' JUST TO RE NEGATIVF)
(YOU ARE BEING A BIT NEGATIVE) (WHY NOT) (WHY 'NO')))
(MY = YOUR 2 ((0 YOUR 0 (/FAMILY) 0)
(TELL ME MORE ABOUT YOUR FAMILY) ( WHO ELSE IN YOUR FAMILY
5) (YOUR 4) (WHAT ELSE COMES TO MIND WHEN YOU THINK OF
YOUR 4))
((0 YOUR 0) (YOUR 3) (WHY DO YOU SAY YOUR 3) (DOES THAT
SUGGEST ANYTHING ELSE WHICH BELONGS TO YOU) (IS IT
IMPORTANT TO YOU THAT 2 3)))
(CAN ((0 CAN I 0)  (YOU BELIEVF I CAN 4 DON'T YOU) (=WHAT)
(YOU WANT ME TO BE ABLE TO 4) (PERHAPS YOU WOULD LIKF TO BF
ABLE TO 4 YOURSELF)) ((0 CAN YOU 0) (WHETHER OR NOT YOU CAN 4
DEPENDS ON YOU MORE THAN ON ME) (DO YOU WANT TO BE ABLE TO 4)
(PERHAPS YOU DON'T WANT TO 4) (=WHAT)))
(WHAT ((0) (WHY DO YOU ASK) (DOES THAT QUESTION INTFRFST YOU)
(WHAT IS IT YOU REALLY WANT TO KNOW)
(ARE SUCH QUESTIONS MUCH ON YOUR MIND)
(WHAT ANSWER WOULD PLEASE YOU MOST)
(WHAT DO YOU THINK) (WHAT COMES TO YOUR MIND WHEN YOU ASK THAT)
(HAVE YOU ASKED SUCH QUESTION BFFORF)
(HAVE YOU ASKED ANYONE ELSE)))
(BECAUSE ((0) (IS THAT THE REAL RFASON) (DON'T ANY OTHER RFASONS
COME TO MIND) (DOES THAT REASON SEEM TO EXPLAIN ANYTHING ELSF)
(WHAT OTHER REASONS MIGHT THERE BF)))
(WHY ((0 WHY DON'T I 0) (DO YOU BELIEVF I DON'T 5) (PERHAPS I
WILL 5 IN GOOD TIME) (SHOULD YOU 5 YOURSELF) (YOU WANT ME TO 5)
(=WHAT))
(=WHAT))
((0 WHY CAN'T YOU 0) (DO YOU THINK YOU SHOULD BF ABLF TO 5)
(DO YOU WANT TO BE ABLE TO 5) (DO YOU BFLIFVF THIS WILL HFLP YOU
TO 5) (HAVE YOU ANY IDEA WHY YOU CAN'T 5) (=WHAT))
(= WHAT))
(EVERYONE 2 ((0 (* EVERYONE EVERYBODY NOBODY NOONF) 0 )
(REALLY, 2) (SURELY NOT 2) (CAN YOU THINK OF
ANYONE IN PARTICULAR) (WHO, FOR EXAMPLE) (YOU ARE THINKING OF
A VERY SPECIAL PERSON)
(WHO, MAY I ASK) (SOMEONE SPECIAL PERHAPS)
(YOU HAVE A PARTICULAR PERSON IN MIND, DON'T YOU) (WHO DO YOU
THINK YOU'RE TALKING ABOUT)))
(EVERYBODY 2 (= EVERYONE))
(NOBODY 2 (=EVERYONE))
(NOONE 2 (=EVERYONE))
(ALWAYS 1 ((0) (CAN YOU THINK OF A SPECIFIC EXAMPLE) (WHEN)
(WHAT INCIDENT ARE YOU THINKING OF) (REALLY, ALWAYS)))
(LIKE 10 ((0 (*AM IS ARE WAS) 0 LIKE 0) (=DIT))
((0) (NEWKEY)) )
(DIT ((0) (IN WHAT WAY)  (WHAT RESEMBLANCE DO YOU SEE)
(WHAT DOES THAT SIMILARITY SUGGEST TO YOU)
(WHAT OTHER CONNECTIONS DO YOU SEE)
(WHAT DO YOU SUPPOSE THAT RESEMBLANCE MEANS)
(WHAT IS THE CONNECTION, DO YOU SUPPOSE)
(COULD THERE REALLY BE SOME CONNECTION)
(HOW)))
()
```

RECEIVED SEPTEMBER, 1965

Computational Linguistics

D. G. BOBROW, Editor

Contextual Understanding by Computers

JOSEPH WEIZENBAUM
Massachusetts Institute of Technology, Cambridge, Mass.

A further development of a computer program (ELIZA) capable of conversing in natural language is discussed. The importance of context to both human and machine understanding is stressed. It is argued that the adequacy of the level of understanding achieved in a particular conversation depends on the purpose of that conversation, and that absolute understanding on the part of either humans or machines is impossible.

We are here concerned with the recognition of semantic patterns in text.

I compose my sentences and paragraphs in the belief that I shall be understood—perhaps even that what I write here will prove persuasive. For this faith to be at all meaningful, I must hypothesize at least one reader other than myself. I speak of *understanding*. What I must suppose is clearly that my reader will recognize patterns in these sentences and, on the basis of this recognition, be able to recreate my present thought for himself. Notice the very structure of the word "recognize," that is, know again! I also use the word "recreate." This suggests that the reader is an active participant in the two-person communication. He brings something of himself to it. His understanding is a function of that something as well as of what is written here. I will return to this point later.

Much of the motivation for the work discussed here derives from attempts to program a computer to understand what a human might say to it. Lest it be misunderstood, let me state right away that the input to the computer is in the form of typewritten messages—certainly not human speech. This restriction has the effect of establishing a narrower channel of communication than that available to humans in face-to-face conversations. In the latter, many ideas that potentially aid understanding are communicated by gestures, intonations, pauses, and so on. All of these are unavailable to readers of telegrams—be they computers or humans.

Further, what I wish to report here should not be confused with what is generally called content analysis. In the present situation we are concerned with the fragments of natural language that occur in conversations, not with complete texts. Consequently, we cannot rely on the texts we are analyzing to be grammatically complete or correct. Hence, no theory that depends on parsing of presumably well-formed sentences can be of much help. We must depend on heuristics and other such impure devices instead.

The first program to which I wish to call attention is a particular member of a family of programs which has come to be known as DOCTOR. The family name of these programs is ELIZA. This name was chosen because these programs, like the Eliza of Pygmalion fame, can be taught to speak increasingly well. DOCTOR causes ELIZA to respond roughly as would certain psychotherapists (Rogerians). ELIZA performs best when its human correspondent is initially instructed to "talk" to it, via the typewriter, of course, just as one would to a psychiatrist.

I chose this mode of conversation because the psychiatric interview is one of the few examples of categorized dyadic natural language communication in which one of the participating pair is free to assume the pose of knowing almost nothing of the real world. If, for example, one were to tell a psychiatrist "I went for a boat ride" and he responded "Tell me about boats," one would not assume that he knew nothing about boats, but that he had some purpose in so directing the subsequent conversation. It is important to note that this assumption is one made by the speaker. Whether it is realistic or not is an altogether separate question. In any case, it has a crucial psychological utility in that it serves the speaker to maintain his sense of being heard and understood. The speaker further defends his impression (which even in real life may be illusory) by attributing to his conversational partner all sorts of background knowledge, insights and reasoning ability. But again, these are the speaker's contribution to the conversation. They manifest themselves inferentially in the interpretations he makes of the offered responses. From the purely technical programming point of view, the psychiatric interview has the advantage that it eliminates the need for storing explicit information about the real world.

The human speaker will contribute much to clothe ELIZA's responses in vestments of plausibility. However, he will not defend his illusion (that he is being understood) against all odds. In human conversation a speaker will

Work reported herein was supported (in part) by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

make certain (perhaps generous) assumptions about his conversational partner. As long as it remains possible to interpret the latter's responses to be consistent with those assumptions, the speaker's image of his partner remains undamaged. Responses which are difficult to so interpret may well result in an enhancement of the image of the partner, by additional rationalizations which then make more complicated interpretations of his responses reasonable. When, however, such rationalizations become too massive and even self-contradictory, the entire image may crumble and be replaced by another ("He is not, after all, as smart as I thought he was".) When the conversational partner is a machine (the distinction between machine and program is not useful here), the idea of credibility may well be substituted for that of plausibility in the above.

"When in conversation with ELIZA, the user types in some statement or set of statements in natural language using normal punctuation and sentence structures."[1] "The user's statement is terminated by a double carriage return which serves to turn control over to ELIZA. ELIZA then analyzes the user's statement and generates some response which it types out. Control is then again with the user.

"A typical conversation is the following:

```
Men are all alike.
IN WHAT WAY
They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE
Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
HE says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
It's true. I am unhappy.
DO YOU THINK COMING HERE WILL HELP YOU NOT TO
  BE UNHAPPY
I need some help, that much seems certain.
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
Perhaps I could learn to get along with my mother.
TELL ME MORE ABOUT YOUR FAMILY
My mother takes care of me.
WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
My father.
YOUR FATHER
WHAT RESEMBLANCE DO YOU SEE
You are not very aggressive but I think you don't want me to
  notice that.
WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE
You don't argue with me.
WHY DO YOU THINK I DON'T ARGUE WITH YOU
You are afraid of me.
DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
My father is afraid of everybody.
WHAT ELSE COMES TO MIND WHEN YOU THINK OF
  YOUR FATHER
Bullies.
DOES THAT HAVE ANYTHING TO DO WITH THE FACT
  THAT YOUR BOYFRIEND MADE YOU COME HERE
```

"The capitalized lines are the machine responses.

I described the original ELIZA program in some detail in [6]. For the present it is sufficient to recall that ELIZA's conversational powers are governed by a *script*. A script consists of a set of key words and key phrases each of which is associated with a set of so-called decomposition and reassembly rules. A decomposition rule is a data structure which searches a text for specified patterns, and, if such patterns are found, decomposes the text into disjoint constituents. A reassembly rule is a specification for the construction of a new text by means of recombinations of old and possible addition of new constituents. From one point of view, an ELIZA script is a program and ELIZA itself an interpreter. From another perspective, ELIZA appears as an actor who must depend on a script for his lines. The script determines the contextual framework within which ELIZA may be expected to converse plausibly.

"The gross procedure of the program is quite simple; the text is read and inspected for the presence of a key word. If such a word is found, the sentence is transformed according to a rule associated with the key word; if not, a content-free remark or, under certain conditions, an earlier transformation is retrieved. A rule-cycling mechanism delays repetition of responses to identical keys as long as possible. The text so computed or retrieved is then printed out."[1]

One of the principle aims of the DOCTOR program is to keep the conversation going—even at the price of having to conceal any misunderstandings on its own part. We shall see how more ambitious objectives are realized subsequently. In the meanwhile, the above discussion already provides a framework within which a number of useful points may be illuminated.

By far the most important of these relates to the crucial role *context* plays in all conversations. The subject who is about to engage in his first conversation with the DOCTOR is told to put himself in a role-playing frame of mind. He is to imagine that he has some problem of the kind one might normally discuss with a psychiatrist, to pretend he is actually conversing with a psychiatrist, and under no circumstances to deviate from that role. While some of the responses produced by the program are not very spectacular even when the subject follows his instructions, it is remarkable how quickly they deteriorate when he leaves his role. In this respect, the program mirrors life. Real two-person conversations also degenerate when the contextual assumptions one participant is making with respect to his partner's statements cease to be valid. This phenomenon is, for example, the basis on which many comedies of error are built.

These remarks are about the *global* context in which the conversation takes place. No understanding is possible in the absence of an established global context. To be sure, strangers do meet, converse, and immediately understand one another (or at least believe they do). But they operate in a shared culture—provided partially by the very language they speak—and, under any but the most trivial circumstances, engage in a kind of hunting behavior which has as its object the creation of a contextual framework. Conversation flows smoothly only after these preliminaries are completed. The situation is no different with respect to visual pattern recognition—a visual pattern may appear utterly senseless until a context within which it may be recognized (known again, i.e., understood) is provided. Very often, of course, a solitary observer arrives at an appropriate context by forming and testing a number of hypotheses. He may later discover that the pattern he "recognized" was not the one he was intended to "see," i.e., that he hypothesized the "wrong" context. He may see the "correct" pattern when given the "correct" context. It doesn't mean much to say that the pattern "is" such and such. We might, for example, find a string of Chinese characters beautiful as long as we don't know what they spell. This, an apparent impoverishment, i.e., really a broadening, of context will enhance the esthetic appeal of a pattern. Similarly, many people think anything said in French is charming and romantic precisely *because* they don't understand the language.

---

[1] The cooperation of the editors of the *Communications of the ACM*, in permitting the extensive quotations from the paper "ELIZA," Vol. 9, No. 1, January, 1966, by the author is hereby gratefully acknowledged.

In real conversations, global context assigns meaning to what is being said in only the most general way. The conversation proceeds by establishing subcontexts, sub-subcontexts within these, and so on. It generates and, so to speak, traverses a contextual tree. Beginning with the topmost or initial node, a new node representing a subcontext is generated, and from this one a new node still, and so on to many levels. Occasionally the currently regnant node is abandoned—i.e., the conversation ascends to a previously established node, perhaps skipping many intermediate ones in the process. New branches are established and old ones abandoned. It is my conjecture that an analysis of the pattern traced by a given conversation through such a directed graph may yield a measure of what one might call the consequential richness of the conversation. Cocktail party chatter, for example, has a rather straight line character. Context is constantly being changed—there is considerable chaining of nodes—but there is hardly any reversal of direction along already established structure. The conversation is inconsequential in that nothing being said has any effect on any questions raised on a higher level. Contrast this with a discussion between, say, two physicists trying to come to understand the results of some experiment. Their conversation tree would be not only deep but broad as well, i.e., they would ascend to an earlier contextual level in order to generate new nodes from there. The signal that their conversation terminated successfully might well be that they ascended (back to) the original node, i.e., that they are again talking about what they started to discuss.

For an individual the analog of a conversation tree is what the social psychologist Abelson calls a *belief structure*. In some areas of the individual's intellectual life, this structure may be highly logically organized—at least up to a point; for example, in the area of his own profession. In more emotionally loaded areas, the structure may be very loosely organized and even contain many contradictions. When a person enters a conversation he brings his belief structures with him as a kind of agenda.

A person's belief structure is a product of his entire life experience. All people have some common formative experiences, e.g., they were all born of mothers. There is consequently some basis of understanding between any two humans simply because they are human. But, even humans living in the same culture will have difficulty in understanding one another where their respective lives differed radically. Since, in the last analysis, each of our lives is unique, there is a limit to what we can bring another person to understand. There is an ultimate privacy about each of us that absolutely precludes full communication of any of our ideas to the universe outside ourselves and which thus isolates each one of us from every other noetic object in the world.

There can be no total understanding and no absolutely reliable test of understanding.

To know with certainty that a person understood what has been said to him is to perceive his entire belief structure and *that* is equivalent to sharing his entire life experience. It is precisely barriers of this kind that artists, especially poets, struggle against.

This issue must be confronted if there is to be any agreement as to what machine "understanding" might mean. What the above argument is intended to make clear is that it is too much to insist that a machine understands a sentence (or a symphony or a poem) only if that sentence invokes the same imagery in the machine as was present in the speaker of the sentence at the time he uttered it. For by that criterion no human understands any other human. Yet, we agree that humans do understand one another to *within acceptable tolerances*. The operative word is "acceptable" for it implies *purpose*. When, therefore, we speak of a machine understanding, we must mean understanding as limited by some objective. He who asserts that there are certain ideas no machines will ever understand can mean at most that the machine will not understand these ideas tolerably well because they relate to objectives that are, in his judgement, inappropriate with respect to machines. Of course, the machine can still deal with such ideas symbolically, i.e., in ways which are reflections—however pale—of the ways organisms for which such objectives are appropriate deal with them. In such cases the machine is no more handicapped than I am, being a man, in trying to understand, say, female jealousy.

A two-person conversation may be said to click along as long as both participants keep discovering (in the sense of uncovering) identical nodes in their respective belief structures. Under such circumstances the conversation tree is merely a set of linearly connected nodes corresponding to the commonly held parts of the participants' belief structures. If such a conversation is interesting to either participant, it is probably because the part of the belief structure being made explicit has not been consciously verbalized before, or has never before been attached to the higher level node to which it is then coupled in that conversation, i.e., seen in that context, or because of the implicit support it is getting by being found to coexist in someone else.

Backtracking over the conversation tree takes place when a new context is introduced and an attempt is made to integrate it into the ongoing conversation, or when a new connection between the present and a previous context is suggested. In either case, there is a need to reorganize the conversation tree. Clearly the kind of psychotherapist initiated by the DOCTOR program restricts himself to pointing out new connectivity opportunities to his patients. I suppose his hope is that any reorganization of the conversation tree generated in the therapy session will ultimately reflect itself in corresponding modifications of his patients' belief structures.

I now turn back to the program reproduced earlier. I hope the reader found the conversation quoted there to be smooth and natural. If he did, he has gone a long way toward verifying what I said earlier about the investment a human will make in a conversation. Any continuity the reader may have perceived in that dialogue—excepting only the last machine response—is entirely illusionary. A careful analysis will reveal that each machine response is a response to the just previous subject input. Again with the exception of the last sentence, the above quoted conversation has no subcontextual structure at all. Nor does the description of the program given in [6] give any clues as to how subcontexts might be recognized or established or maintained by the machine.

To get at the subcontext issue, I want to restate the overall strategy in terms somewhat different from those used above. We may think of the ELIZA script as establishing the global context in which subsequent conversations will be played out. The DOCTOR script establishes the Rogerian psychiatric context. The key word listings are guesses about what significant words might appear in the subject's utterances. The appearance of any of these words in an actual input tentatively directs the system's attention to decomposition rules associated with the identified word. These rules are hypotheses about what an input containing such words might mean. An important property of the system is that just as the entire set of key words need not be searched in the analysis of every input sentence, neither does the whole list of hypotheses about what *any* input might mean have to be tested in order to generate a response to a specific sentence. Should the set of hypotheses associated with the currently regnant key word be exhausted without success, there is a way (although not here described) to go to decomposition rules associated with lower ranking key words that may have appeared in the input string.

Much is known about an input sentence once it matches a decomposition rule. If, for example, the rule

(O MY (/FAMILY) DRINKS O)

matches a sentence, then we know that the subject matter is a family member, and that this person drinks. (The notation "(/ A B C)" means that in that position a word "tagged" with any of the symbols A or B or C is looked for. Words are tagged as part of the script. This mechanism is not further discussed here.) A reassembly rule can subsequently identify the family member in question, e.g., father. The successful matching of a decomposition rule—hence, the verification of a hypothesis—is therefore an opportunity not only to generate a plausible response but also to establish a subcontextual node. The laying down of such a node does not mean merely remembering what is now being talked about, but what is more important, making hypotheses about what is likely to be said next within the new contextual framework.

The realization of such a mechanism in an ELIZA script involves attaching a substructure to decomposition rules which says, in effect, that if a so modified decomposition rule matches an input sentence and a suitable subcontext exists, then control is transferred to a set of decomposition and reassembly rules appropriate to that subcontext. The details of mechanization are not important here. I will, however, show its effect by exhibiting the following conversational fragment from the DOCTOR program:

Because you are so stubborn.
IS THAT THE REAL REASON
Maybe
WHAT IS A BETTER REASON THEN
My stomach hurts.
TELL ME ABOUT YOUR PAINS IN GENERAL
My head bothers me, too.
YOU SEE SOME PSYCHOLOGICAL REASON WHY YOUR HEAD SHOULD PAIN YOU
Maybe.
YOU THINK COMING HERE MIGHT HELP ALLEVIATE YOUR HEAD PAIN
Certainly.
LET'S TALK ABOUT YOUR FEELINGS WHEN YOU THINK OF YOUR HEAD AND ITS PAINS

We are here in the middle of a dialog. The first sentence is an input to the program, the next a machine response, and so on in alternation.

The word "because" was the key that triggered the first machine response. At the same time the *reason* subcontext, so to speak, was established. Notice that the next subject input, i.e., the word "maybe", was interpreted as meaning "no," as is appropriate in that particular context, and that a contextually correct machine response was produced. The next input serves to establish a "hurt" subcontext. I believe the conversation is self-explanatory from that point on. Notice, however, that in the new subcontext the word "maybe" which was earlier interpreted to mean "no" is now seen as meaning "yes." Under still other circumstances it would be understood as indicating uncertainty.

My secretary watched me work on this program over a long period of time. One day she asked to be permitted to talk with the system. Of course, she knew she was talking to a machine. Yet, after I watched her type in a few sentences she turned to me and said "Would you mind leaving the room, please?" I believe this anecdote testifies to the success with which the program maintains the illusion of understanding. However, it does so, as I've already said, at the price of concealing its own misunderstandings. We all do this now and then, perhaps in the service of politeness or for other reasons. But we cannot afford to elevate this occasional tactic to a universal strategy. Thus, while the DOCTOR program may be useful as an instrument for the analysis of two-person conversations, and while it is certainly fun, its aim must be changed from that of concealment of misunderstanding to its explication.

Another difficulty with the system currently under discussion is that it can do very little other than generate plausible responses. To be sure, there are facilities for keeping and testing various tallies as well as other such relatively primitive devices, but the system can do no generalized computation in either the logical or numerical sense. In order to meet this and other deficiencies of the original ELIZA system, I wrote a new program, also called ELIZA, which has now replaced its ancestor.

The ELIZA differs from the old one in two main respects. First, it contains an *evaluator* capable of accepting expressions (programs) of unlimited complexity and evaluating (executing) them. It is, of course, also capable of storing the results of such evaluations for subsequent retrieval and use. Secondly, the idea of the script has been generalized so that now it is possible for the program to contain three different scripts simultaneously and to fetch new scripts from among an unlimited supply stored on a disk storage unit, intercommunication among coexisting scripts is also possible.

The major reason for wishing to have several scripts available in the core (i.e., high speed) memory of the computer derives from the arguments about contexts I made above. The script defines, so to speak, a global context within which all of the subsegment conversation is to be understood. We have seen that it is possible for a single script to establish and maintain subcontexts. But what is a subcontext from one point of view is a major (not to say global) one as seen from another perspective. For example, a conversation may have as its overall framework the health of one of the participants but spend much time under the heading of stomach disorders and headache remedies.

In principle one large, monolithic ELIZA script could deal with this. However, such a script would be very long and extremely difficult to modify and maintain. Besides, long exposure to computer programming should at least instill a healthy respect for subroutines, their power and utility. In the new ELIZA system scripts are in fact very much like subroutines. One script plays the role of the so-called "main program," i.e., it is the one for calling in and replacing subscripts. This is the one which, in an important sense, governs the computer's role in the conversation. Such a script may, for example, play the role of an initial interviewer in a hospital, i.e., a person who knows just enough about the field of medicine and the organization of the hospital to be able to determine, by means of appropriate questions, to whom the incoming patient is to be directed. A more highly specialized script then plays the role of that person. It may call on still further levels of scripts—and each script, except for the "main" one, can return the issue up one or more levels with information "explaining" that action.

We saw earlier how the word "maybe" was differentially interpreted in what might be called different micro contexts. The hierarchy of scripts now under discussion makes much more nearly global contextual distinctions possible. For example, the word "tone" probably means muscle tone to the internist but audio tone to the ear specialist. The occurrence of that word as a key word would establish an entirely different contextual mode, or call in a different subscript, depending on whether it was identified in an internist or an ear specialist script.

Of course, another important advantage of this kind of subscripting is that scripts (sub or otherwise) can presumably be written by specialists in a very narrow field. The fact that scripts now tend to be rather short means that a specialist might actually find the time to produce one as part of a larger cooperative effort.

The conversations reproduced below were governed by a script in the new ELIZA system. This script was designed to illustrate the power of the evaluator to which ELIZA has access. The key word and other such mechanisms are not very different from those described above. The important departure is that expressions can be turned over to the evaluator, evaluated, and the results displayed and stored.

```
The area of a circle is pi times the radius of the circle squared.
The area of a sphere is pi times the diameter of the sphere squared.
The area of a square is the side of the square squared.
The diameter of an object is twice the radius of the object.

The radius of a ball is 10.
What is the area of the ball.
THE ASSUMPTION THAT BALL IS CIRCLE IS CONSIST-
  ENT--SHALL I GO ON?  no
THE ASSUMPTION THAT BALL IS SPHERE AND OBJECT
  IS CONSISTENT--SHALL I GO ON?  yes
IT'S 1256.6359

What is the diameter of the ball.
IT'S 20

The radius of a dish is 10.
What 's the area of the dish.
THE ASSUMPTION THAT DISH IS CIRCLE IS CONSIST-
  ENT--SHALL I GO ON?  yes
IT'S 314.15899

The radius of a globe is 10.
A globe is a sphere. A sphere is an object.
What is the area of the globe.
IT'S 1256.6359

The side of a plate is 10.
What is the area of the plate.
THE ASSUMPTION THAT PLATE IS SQUARE IS CONSIST-
  ENT--SHALL I GO ON?  yes
```

IT'S 100
When I say board I mean plate.
What is 3 times the side of the board.

It's 30.

Please call that x.
Let f(z) be z squared minus z.
Compute f(x).

It's 870.
When I say ich sage I mean I say.
When ich sage meine ich I mean I mean.
Wenn ich sage wieviel ist meine ich what is.
Wenn ich sage quadrat meine ich squared.
Nun, sagen sie mir bitte, wieviel is x quadrat.

I DIDN'T UNDERSTAND YOU.

Ja, tut mir leid, ich sollte fragen wieviel IST x quadrat.

IT'S 900.

Thank you.

YOU'RE QUITE WELCOME, COME AGAIN

The particular example shown here is one in which the system is asked not merely to entertain but to perform some useful service. The script is one which establishes a context in which a few terms can be interpreted as giving directions about the evaluation of certain expressions. The number of keys which actually lead to decomposition rules is very small. They are as follows:

I mean	
Is	(* a an)
Of	(* a an the)
What	(* the is)
Where	
Call	
Let	
Define	
Compute	
=	
Also	
Thanks	

(A key of the form "Is (* a an)" will match either "Is a" or "Is an".) In addition, there are 1/ substitution keys like "Twice = 2*". The effect of that particular one is to substitute the symbols "2*" wherecever the word "twice" appeared in the input text. Of course, the evaluator sitting behind ELIZA, so to speak, recognizes words like "SQRT", "LOG", etc. The function of this script is to interpret the user's wishes with respect to the evaluation of expression, perform certain translation functions on these expressions, and control the traffic between the input/output system of ELIZA and that of the evaluator.

Consider the dozen keys shown above. The sentence "*Let* me try to *define what the call of the* sea means" contains five of these keys. It could perhaps be understood by the DOCTOR but not by the program we are now considering. It would reply "I didn't understand you."

I call attention to this contextual matter once more to underline the thesis that, while a computer program that "understands" natural language in the most general sense is for the present beyond our means, the granting of even a quite broad contextual framework allows us to construct practical language recognition procedures.

The present script is designed to *reveal*, as opposed to *conceal*, lack of understanding and misunderstanding. Notice, for example, that when the program is asked to compute the area of the ball, it doesn't yet know that a ball is a sphere and that when the diameter of the ball needs to be computed the fact that a ball is an object has also not yet been established. Appropriate questions are therefore asked. But when, later on, the area of the globe is required, all the parameters are known and no questions are asked. The transcript also makes clear that the program now has memory and, of course, that it can compute.

I can imagine at least two questions that might have arisen in the mind of the reader in response to the above material. First, but least important, is what role the kind of psychiatric program shown here has in serious psychotherapy. My answer to that must be that psychotherapy is not my field and that I have no intention to pursue any therapeutic implications of this work. It is clear that there is considerable psychological material to be exploited here, material that may conceivably lead to some therapeutic developments. There may also be some use in this kind of technique in simulation of the neurotic processes. K. M. Colby, a practicing psychoanalyst, does propose to use conversational programs as psychotherapeutic tools vis-a-vis real patients. I believe the unsolved technical problems in the computer realm alone are so enormous as to guarantee the postponement of the realization of such a goal for yet a very long time.

The second question concerns itself with the implications of this work on so-called teaching machines. Here I feel on much firmer ground. It is my belief that the kind of programmed teaching that is based on multiple choice testing of the student is best done without the use of computers. With the techniques described here, it becomes possible to instruct the student in a corner of some field and then, the contextual framework having been established, to have the student interact with the machine in natural language.

The following conversation, based on a script written by Dr. Edwin Taylor of the Education Research Center of MIT, makes the point very well.

PLEASE CONSIDER HOW WE MIGHT SYNCHRONIZE TWO CLOCKS TO READ THE 'SAME TIME'. ONE OF THESE CLOCKS IS IN SAN FRANCISCO, THE OTHER IS IN NEW YORK. AS YOU THINK ABOUT THE PROBLEM, ASK ME ANY QUESTIONS THAT OCCUR TO YOU. WHEN YOU ARE READY TO PROPOSE A SYNCHRONIZATION METHOD, TYPE 'READY'. THEN I WILL QUIZ YOU ABOUT YOUR PROPOSED METHOD.

what do you mean by synchronize

FOR THE PRESENT MAKE USE OF THE EVERYDAY MEANINGS OF SUCH PHRASES AS 'SAME TIME' AND 'SYNCHRONIZE'. LATER WE WILL DISCUSS THE MEANING OF THESE TERMS MORE CAREFULLY

may i use telegraph signals

CERTAINLY, FEEL FREE TO USE TELEGRAPH

ready

NOW I WILL QUIZ YOU ON YOUR SYNCHRONIZATION METHOD DOES YOUR METHOD OF SYNCHRONIZATION INVOLVE THE PROPAGATION OF SOME SIGNAL—OR MATERIAL OBJECT—FROM ONE PLACE TO ANOTHER

yes

WHAT KIND OF PROPAGATING SIGNAL OR TRAVELING OBJECT DO YOU PROPOSE TO USE

a watch

WILL YOUR WATCH MOVE AT A SPEED VERY MUCH SMALLER THAN THE SPEED OF LIGHT

of course

GOOD. TRAVELLING CLOCKS CAN BE USED TO SYNCHRONIZE DISTANT CLOCKS TO ARBITRARILY HIGH ACCURACY IF THEY TRAVEL SLOWLY ENOUGH. YOU MAY LEAVE THE PROBLEM BY PUSHING THE'QUIT' BUTTON TWICE. IF YOU WOULD LIKE TO EXPLORE FURTHER THE MATTERS OF PRINCIPLE ASSOCIATED WITH OTHER METHODS, CHOOSE ANOTHER WAY TO COMMUNICATE BETWEEN THE DISTANT CLOCKS
WHAT KIND OF PROPAGATING SIGNAL OR TRAVELING OBJECT DO YOU PROPOSE TO USE

light

WELL, OF COURSE THE LIGHT SIGNAL WILL NOT FOLLOW THE CURVATURE OF THE EARTH. SO LET US ASSUME THAT SOME SYSTEM OF RELAY TOWERS IS USED. NOW LET ME ASK, WILL THE LIGHT SIGNAL TRAVEL DIRECTLY BETWEEN NEW YORK AND SAN FRANCISCO OR WILL THE LIGHT SIGNAL TRAVEL TO OR FROM SOME INTERMEDIATE POINT

it will travel from st louis in both directions

IS YOUR INTERMEDIATE POINT EQUIDISTANT FROM NEW YORK AND SAN FRANCISCO

let's say it is, yes

MY UNDERSTANDING IS THAT YOU EXPECT TO USE TWO SIGNALS AND THAT THESE SIGNALS WILL EITHER CONVERGE ON OR BE SENT OUT FROM A POINT EQUIDISTANT FROM NEW YORK AND SAN FRANCISCO. IS MY UNDERSTANDING ON THIS POINT CORRECT

yes

WILL THIS PAIR OF SIGNALS CONVERGE ON THE INTERMEDIATE POINT OR BE SENT OUT FROM THIS POINT
they will be sent from the point
WILL THE TWO SIGNALS LEAVE THE INTERMEDIATE POINT AT THE SAME TIME

certainly

AHA, I'VE GOT YOU. YOU ARE TRYING TO SYNCHRONIZE THE NEW YORK AND SAN FRANCISCO CLOCKS TO READ THE 'SAME TIME'. HOW CAN IT BE VALID TO ASSUME A 'SAME TIME' FOR THE DEPARTURE OF A PAIR OF SIGNALS FROM THE MIDPOINT

because the pair of signals leave from the same point

RIGHT
THE PHRASE 'SAME TIME' HAS AN UNAMBIGUOUS MEANING WHEN APPLIED TO EVENTS THAT OCCUR NEAR THE SAME POINT IN SPACE. NOW YOU ARE GOING TO CONTINUE YOUR ARGUMENT BY SAYING THAT THE TWO SIGNALS THAT START FROM THE MIDPOINT AT THE SAME TIME WILL PRESUMABLY REACH NEW YORK AND SAN FRANCISCO AT THE SAME TIME. THESE SIGNALS CAN THUS BE USED TO SYNCHRONIZE THE NEW YORK AND SAN FRANCISCO CLOCKS. IS THIS THE WAY YOUR ARGUMENT WILL CONTINUE

yes

ONE FINAL QUESTION. WHAT ASSUMPTION ARE YOU MAKING ABOUT THE SPEEDS OF THE TWO SIGNALS THAT TRAVEL IN OPPOSITE DIRECTIONS
that these speeds are equal
VERY GOOD. YOU HAVE DEVISED A VALID METHOD FOR SYNCHRONIZING TWO CLOCKS, ONE IN NEW YORK AND ONE IN SAN FRANCISCO. I AM NOW RETURNING YOU TO THE CTSS SYSTEM

I have chosen to skew this discussion in the direction of "understanding" and have used that difficult word almost as if it were synonymous with "recognition." Actually, recognition is a prerequisite to understanding. The latter provides a test of the former. Even though no test of understanding can be defined with rigor, we do feel we are understood when the person we're talking to asks what we believe to be relevant questions and when he finally answers our question or solves the problem we posed. By such criteria, the conversations displayed above justify the assertion that the programs that produced the responses recognized textual patterns and understood what they were being told.

Finally, I wish to call attention to three pieces of work, two of which predate the programs discussed here and to whose authors I owe a considerable intellectual debt. The last is more recent but nevertheless highly relevant to my own current line of attack.

The SIR program of Raphael is capable of inferential data acquisition in a way analogous to that displayed in the ELIZA ball and sphere conversation displayed above. Notice that in that conversation the program had to infer that a ball was a sphere and an object. Once that inference was affirmed, the program retained the information by, in this case, associating with ball the fact that it is a sphere and an object and with sphere and object that ball is an instance of each, respectively. SIR is a program which specializes in establishing such relationships, remembering and invoking them when required. One of its principal aims was to establish methodology for formalizing a calculus of relations and even relations among relations.

Bobrow's program STUDENT is capable of solving so-called algebra word problems of the kind that are typically given in high school algebra texts. He uses a mechanism not very different from an ELIZA script. Its chief task is to transform the input text, i.e., the natural language statement of an algebra word problem, into a set of simultaneous linear equations that may then be evaluated to produce the desired result. A particular strength of his program is its power to recognize ambiguities and resolve them, often by appeal to inferentially acquired information but sometimes by asking questions.

The work of Quillian is mainly directed toward establishing data structures capable of searching semantic dictionaries. His system could, for example, decide that the words "work for" in the sentence "John works for Harry." mean "is employed by", while the same words appearing in the sentence "That algorithm works for all even numbers that are not perfect squares." mean "is applicable to."

Each of the computer papers referenced below represents an attack on some component of the machine understanding problem. That problem is not yet solved.

Received April, 1967

REFERENCES

1. Bobrow, D. G. Natural language input for a computer problem-solving system. Ph.D. Thesis. MIT, Dept. of Mathematics, Cambridge, Mass. 1964.
2. Colby, Kenneth Mark. Computer simulation of change in personal belief systems. Paper delivered in Section L_2. The Psychiatric, Sciences, General Systems Research, AAAS Berkeley Meeting, December 29, 1965. To appear in *Behav. Sci.*, 1967.
3. Quillian, M. R. Semantic memory. Ph.D. Thesis, Carnegie Inst. of Technology. Pittsburgh, Pa., 1966.
4. Raphael, B. SIR: A computer program for Semantic Information Retrieval. Ph.D. Thesis, MIT, Dept. of Mathematics, Cambridge, Mass., 1964.
5. Rogers, C. *Client Centered Therapy: Current Practice, Implications and Theory.* Houghton Mifflin, Boston, 1951.
6. Weizenbaum, Joseph. ELIZA a computer program for the study of natural language communication between man and machine. *Comm. ACM 9*, 1 (Jan. 1966), 36-45.

APPENDIX C

TRAC Functions in GLURP

APPENDIX C

TRAC FUNCTIONS IN GLURP

#(AD:D1:D2:Z)
Adds decimal numeric digits D1 to digits D2. A negative number is indicated by a leading minus sign as in "-3". No decimal points may be included in the number. If the arithmetic capacity of the machine is exceeded, the value of the string is Z. Normally, the value of the string is D1 plus D2. Does not change forms storage.

#(AN:N:D:Z) ADD AND STORE
This function is similar to AD, except that the first argument is a string name instead of a number. The contents of N are incremented by D. If the arithmetic capacity of the machine is exceeded, the value of the string is Z and forms storage remains unchanged. Normally, the value of the string is the contents of N plus D, and this value replaces the previous contents of N.

#(BA:N1:N2) BOOLEAN AND
This function performs a Boolean "and" function of the Boolean string N2 and the Boolean contents of string N1. The results are left in N1 and the function has no value. For every position that is a 1 both in N1 and the contents of N2, a 1 is placed in that position of N1.

#(BC:N) BOOLEAN COMPLEMENT
This function reverses the sense of each position in Boolean string contained in N. Every 1 is changed to 0, every 0 is changed to 1. The result is stored in N. The function has no replacement value. In our version of BC, the number of bits is always a multiple of 8.

#(BE:N1:N2) BOOLEAN EXCLUSIVE OR
Performs a Boolean "exclusive or" function with the contents of N1 and Boolean string N2. Every position that is a 1 in one string and a 0 in the other is set to 1 in N1. The results are stored in N1 and the function has no replacement value.

#(BO:N1:N2) BOOLEAN OR
Performs Boolean "or" function with the contents of N1 and the Boolean string N2. Whenever a position has a 1 in either string, or both strings, a 1 is put in that position in string N1. The function has no replacement value.

#(CB:N) COUNT BINARY BITS
The value is the number of 1's in Boolean string named N. Forms storage is unchanged.

#(CC:N:Z) CALL CHARACTER
The value is the character under the form pointer. Every string has a counter that shows what part of a string is to be considered the current beginning. In this way the programmer can use pieces of a string, letting the machine keep track of where he left off. This is one of the functions

that uses the string in this fashion. If, say, we had string N with contents

abcdefghijkl

the first use of the CC function would find the form pointer pointing at "a". The value of the function, then, would be set as "a" and the form pointer moved up to position "b". The next use of the CC instruction would produce the value of "b", unless some other form-pointer using function was used in between, of course. The CC function skips over segment gaps (see SS function). If, in the example, the forms pointer were pointing to "l", the next use of CC would produce the value "l", but the next use of the function would produce the value Z.

#(CE:N1:N2:Z:N3) COMPARE FOR EQUALITIES

In GLURP strings take on several forms. One of these is the word list, N, consists of a series of words separated by blanks. Each word in the string is considered an item of the list as is every comma.
This function compares word list N1 abainst word list N2. If all the words on N1 are on N2, the value of the function is Z and N3 remains unchanged in form storage. If all the items on N1 are not on N2, then those items which are on both N1 and N2 are put in list N3. Any previous definition of N3 is deleted. The function normally has no replacement value.

#(CI:N1:N2:Z:N3) COMPARE FOR INEQUALITIES

Similar to CE, except that those items which are on list N1 but not on list N2 are put on list N3. The function normally has no replacement value.

#(CK) CALL KEYBOARD

The value of the function is the last student response to a KEYBOARD statement. Forms storage remains unchanged.

#(CL:N:X1:X2:.....) CALL STRING

The value of this function is created by bringing in string N and filling in the segments gaps of value 1 with X1, segment gaps 2 with X2, etc. (see SS function.) Forms storage remains unchanged. In practice, this function is omitted and a "call" is accomplished by using a string name in the function position instead of a function name

#(N:X1:X2:.....)

The colons for segment gaps must be present even if null arguments are presented.

#(CN:N:D:Z) CALL N CHARACTERS

This is similar to CC, except that its value is the next D characters of string N. If D characters do not remain on the string, the value is Z and the form is unchanged. The forms pointer normally is moved to the letter after the D specified characters of the string.

#(CO:N1:N2:N3:N4) CONCATENATE
This function creates a new string N1 which consists of the three string N2, N3, N4 put one next to the other. If one or more arguments are omitted, only those strings specified are used to create the new string, in the order specified. Any previous definition of N1 is deleted. The function has no replacement value. Strings N2, N3, and N4 are left unchanged.

#(CP:D1:D2:Z1:Z2:Z3) ARITHMETIC COMPARE
Compares digits D1 with D2. If D1 is less than C2, the value of the function is Z1; if D1 is the dame as D2, the value of the function is Z2; if D1 is greater than D2, the value is Z3. Forms storage is left unchanged. A null value is treated as zero.

#(CR:N) CALL RESTORE
Restores the form pointer back to the beginning of a string after it has been moved up by CC, CS, PT, etc. The function has no replacement value.

#(CS:N:Z) CALL SEGMENT
The value of this function is the string from the current position of the form pointer to the next segment gap of string N. The value is the rest of the string is no segment gaps remain. If the string is empty, the value is Z. The form pointer is moved to the first character following the segment gap.

#(DC:FILE:ATTRIBUTE:N:Z) DECODE BINARY
The binary string named N is decoded according to the same table that would be used for ATTRIBUTE in FILE. Entries corresponding to the different bits are separated by commas. This string is the value of the function. Only 256 characters are allowed each time DC is used. If there are no more bits to decode, or the decode table is not found, the value is Z. Issuing the same function again causes decoding to continue where it left off.

#(DD:N1:N2:...) DELETE DEFINITION
Deletes strings N1, N2, etc. The function has no replacement value.

#(DE:FILE:ATTRIBUTE:N:Z) DECODE ITEM
The function is replaced by the decoded value of the contents of N. The decoding takes place as if the contents of N were the value of the attribute. If the decoding cannot take place, the value of the function is Z. Forms storage is unchanged.

#(DF:FILE:RECORD:ATTRIBUTE:N:Z) DELETE FROM DISC WITH FETCH
Deletes the data item specified by FILE, RECORD, ATTRIBUTE, from the disc, at the same time inserting it in string N. If N is not specified, the value of the function is the deleted data; otherwide the function has no replacement value. If the data cannot be found, the value of the function is Z.

#(DI:N:X1:X2:....) DELETE ITEMS FROM LIST
In the list named N, items X1, X2, etc. are deleted. The function has no replacement value.

#(DN:N:D:Z) DIVIDE AND STORE
The contents of N are divided by D. In the arithmetic capacity of the machine is exceeded, the value of the string is Z and forms storage is unchanged. Normally, the value of the string is the contents of N divided by D rounded down to the next lower whole number, and this value replaces the previous contents of N.

#(DS:N:X) DEFINE STRING
Defines in forms storage a string named N with contents X. Function has no replacement value.

#(DT:FILE:RECORD:ATTRIBUTE::Z) DELETE FROM DISC
Deletes specified data item from disc. If record cannot be found or is not to be deleted because of its protection key, the value of the function is Z. Otherwise, the function has no replacement value. Forms storage is unchanged.

#(DV:D1:D2:Z) DIVIDE
The value of the function is the quotient of D1 and D2, to the next lower integer. If the arithmetic capacity of the machine is exceeded, the value of the string is Z. Forms storage is unchanged.

#(EQ:X1:X2:Z1:Z2) CHARACTER EQUALITY TEST
Compares string X1 with string X2. If they are equal, the value of the function is Z1, otherwise the value is Z2. Forms storage is unchanged.

#(FO:FILE:RECORD:ATTRIBUTE:N:Z) FETCH ONLY
Gets the specified data item from the disc. If N is specified, the data is put in string N and the function has no replacement value. If N is not specified, the function value is the data item; if the data item cannot be found, the value of the function is Z. The item is not decoded.

#(FR:FILE:RECORD:ATTRIBUTE:N:Z) FETCH RECORD WITH DECODING
Gets the specified data item from the disc. If N is specified, the data is put in string N and the function has no replacement value. If N is not specified, the function value is the data item; if the data item cannot be found, the value of the function is Z. Item is decoded according to the appropriate table. If no deciding is specified, the literal value is returned.

#(GO:STEPNO) GOTO
Transfers script control to statement STEPNO. STEPNO must be in the format DDD.DD where all the D's are digits. This is different from the Z format, in that there is no leading *. The function has no replacement value and forms storage is untouched. Control is transferred immediately as soon as this function is recognized; any remaining functions in the current statement are ignored.

#(ID:FILE:RECORD:ATTRIBUTE:X:Z) INSERT ON DISC
Inserts data item X on the disc with the specified labels. If the label is already represented on the disc, the item is not inserted and the value of the function is Z; otherwise the function has no replacement value. Forms storage is unchanged.

#(IN:N:X:Z) INITIAL
Starting from the form pointer, the form named N is searched for the first location where X matches the string. The value of the function is the string from the pointer up to the character just before the matching string section. If a match is not found, the value is Z. The form pointer is moved to the character following the matching substring, or is not moved if there is no match.

#(KS:D) KEYBOARD SEGMENT
The value of this function is the Dth segment of the last user response as interpreted by the last used DECOMP statement. The keyboard buffer and form storage are unchanged.

#(LC) LOG CRT MESSAGES
Causes messages to the console to be logged on the log tape. This is not the normal mode. There is no replacement value and forms storage is unchanged.

#(LG:X) LOG MESSAGE
The string X is written to the logtape. There is no replacement value and forms storage is unchanged.

#(LI:X:Z) LOGIN
The session is initialized for inquirer X. If X is an unacceptable identification, the value is Z. Forms storage contains NICK, the inquirer's nickname and PASSWORD, the inquirer's password. Normally there is no replacement value.

#(LK:N) LOAD KEYBOARD
The contents of the string N is placed in the keyboard buffer, and can then be used for DECOMP statements. Forms storage is unchanged. The function has no replacement value.

#(LN:X) LIST NAMES
The value is a list of all the names of forms in forms storage separated by the string X. Forms storage is unchanged.

#(LO) LOGOUT
The session is terminated for the inquirer in question. All forms are deleted. There is no replacement value.

#(ML:D1:D2:Z) MULTIPLY
Multiplies decimal digits D1 and D2. No decimal points are allowed. Normally the value is the product of D1 and D2. If the arithmetic capacity of the machine is exceeded, the value is Z. Forms storage is unchanged.

#(MN:N:D:Z) MULTIPLY AND STORE
This function is similar to ML, except that the first argument is a string name instead of a number. The contents of N are multiplied by D. If the arithmetic capacity of the machine is exceeded, the value of the string is Z and forms storage remains unchanged. Normally, the value of the string is the contents of N times D, and this value replaces the previous contents of N.

#(NB:N:X) NEW BOTTOM LIST
Puts item X on the bottom of the list named N. The function has no replacement value.

#(NC) DO NOT LOG CRT MESSAGES
Turns off the logging of CRT messages to the log tape. This is the normal mode. There is no replacement value and forms storage is unchanged.

#(NT:N:X) NEW TOP LIST
Puts item X on the top of the list named N. If the form pointer has been advanced by the use of CC, CS, CN, or PT, only the remaining portion of the list will be used with X at the top. The function has no replacement value.

#(PC:X) PRINT CONTINUOUS
The string X is displayed at the next available space. There is no replacement value, and forms storage is unchanged.

#(PF:N) PRINT FORM
Prints the contents of the string named N, including indications of any segment gaps. The function has no replacement value and forms storage remains unchanged.

#(PL:W:X) PRINT LIST
Prints each word in word list W separated by the character(s) X. The function has no replacement value and forms storage is unchanged.

#(PS:X) PRINT STRING
Prints string X on the next available line. The function has no replacement value and forms storage remains unchanged.

#(PT:N:Z) POP TOP LIST
The value of the function is the next item from the top of word list N. If no words remain, the value is Z. The form pointer is advanced to the next item in the list. Multiple spaces are treated as one space.

#(RD:FILE:RECORD:ATTRIBUTE:X:Z) REPLACE ON DISC
Replaces the data item specified by data X. If the existing item cannot be found, no replacement takes place and the value of the function is Z. Otherwise, the function has no replacement value. Forms storage remains unchanged.

#(RH:FILE:RECORD:Z) READ HOME PRIORITY
See HASM/DASM write-up for description and use of Home Priorities. Function has replacement value of home priority of file and record stated or, if record not found, it has a value of Z. Forms storage is unchanged.

#(RT:STEPNO) RESET STEP
Sets the script branch control at STEPNO, but does not transfer control until the rest of the current statement is analyzed. STEPNO must be in the format DDD.DD where all the D's are digits. There is no leading * as there is in the Z format.

#(SC)
SCRIPT CURRENT
The script name and step number are displayed on the screen. Forms storage is unchanged, and there is no replacement value.

#(SD:X)
SELECTIVE DELETE
Deletes from form storage all strings whose names begin with the characters X. The function has no replacement value.

#(SH:FILE:RECORD:D:Z)
STORE HOME PRIORITY
See HASM/DASM write-up for description and use of home priorities. Function stores D as home priority in stated file and record. Has no replacement value unless the file or record is not found, in which case the value is Z.

#(SN:N:D:Z)
SUBTRACT AND STORE
Subtracts D from the contents of string N and replaces the previous contents of N with this sum. The value of the function is the new contents of N. If the arithmetic capacity of the machine is exceeded, the value of the function is Z and forms storage is unchanged.

#(SR:FILE:RECORD:ATTRIBUTE:X:Z)
STORE RECORD ON DISC
Stores data X on disc with specified label. If label already exists on disc, the old one is deleted and the new item put on instead. The function has no replacement value and forms storage is unchanged.

#(SS:N:X1:X2:...)
SEGMENT STRING
The string named N is scanned from left to right with respect to string X1. If a substring is found matching X1, that substring is removed from string N and marked with a segment marker of value 1. The marker is called a "segment gap". The rest of the string N is scanned with respect to X1 to create any additional segment gaps. The parts of the string not taken by segment gaps are now scanned with respect to X2, and the matching substrings, if any, are marked as segment gaps with value ", etc. The untouched portions of the resulting strings are called "segments".

#(SU:D1:D2:Z)
SUBTRACT
The value of the function is the result of subtracting the number D2 from D1. If the result exceeds the arithmetic capacity of the machine, the value of the function is Z. Forms storage remains unchanged.

#(TB:N:D:Z1:Z2)
TEST BINARY BIT
Tests the Dth binary bit in the Boolean string named N. If that bit is "on", or a 1, the value is Z1; otherwise it is Z2.

#(TF:N:D1:D2:D3:...)
TURN OFF BINARY BIT
Forces the D1, D2, D3... positions of Boolean string named N to be 0. The function has no replacement value.

#(TO:N:D1:D2:D3...)
TURN ON BINARY BIT
Forces the D1, D2, D3... positions of Boolean string named N to be 1. The function has no replacement value.

#(XQ:SCRIPTNAME:STEPNO) EXECUTE

Sends control to script SCRIPTNAME, statement STEPNO. If STEPNO is not designated control goes to the first statement of that script. Control is transferred immediately, without further analysis of the current statement. Forms storage is left unchanged. If the step or script does not exist, control is passed to the script on top of the LINK list.

#(YZ) TRACE DECOMP STATEMENTS

Causes each match attempted in the evaluation of a DECOMP statement to be logged on the LOG tape. This is to be used only in desperation, as it is extremely wordy.

#(ZZ) TURN OFF DECOMP TRACE

Turns off "YZ" or "panic" mode.

APPENDIX D

Permanent Forms

APPENDIX D

PERMANENT FORMS

#(AORAN:X) A OR AN

The value of the function is "AN X" if the first character of X is a, e, i, o, or u. Otherwise the value is "A X".

Any previous definition of A*** is deleted from forms storage. The form is as follows: (@1, @2, @3 indicate segment gaps.)

```
#(DS:A***:@1)
#(DS:A***:#(CC:A***))
#(SS:A***:A:E:I:O:U)
#(EQ:#(A***:::::)#(DD:A***)::AN @1:A @1)
```

#(COUNT:N) COUNT CHARACTERS

The value is the number of characters in the string N.

Previous definitions of A*** and B*** are deleted from forms storage. The form pointer to N is restored to the beginning, although the count is from the pointer to the end of the form. The form is as follows:

```
#(LOOP:A***:100:(#(***:#(CN:@1:10:(#(LOOP:B***:10:(
#(CC:@1:(#(CR:@1)#(COUNT***))))))))))
```

#(COUNT***)

```
#(AD:#(A***)0:#(B***))#(DD:A***:B***)
```

#(ENDINS:X1:X2:X3) END IN S

If the last character of the string X1 is the letter S, the value is X2, otherwise the value is X3. Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(DS:A***@1)
#(***:#(CS:A***))
#(EQ:#(CN:A***:-1):S:@2:@3)#(DD:A***)
```

#(LOOP:N:D:X) LOOP THROUGH

The string X is executed D times. The counter is kept in N in forms storage. If X causes a transfer of control, the value of N will be the number of times the loop was completed.

The form is as follows:

```
#(DS:@1:0)
#(LOOP***:#(AD:@2:1):(@3))
```

LOOP***

```
#(CP:#(AN:@1:1):@2:(@3#(LOOP***:@1:@2:(@3))))
```

#(NUMBER:X:STEPNO1:STEPNO2) NUMBER

If X is all digits, control transfers to STEPNO1, otherwise control transfers to STEPNO2. Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(DS:A***:@1)
#(SS:A***:0:1:2:3:4:5:6:7:8:9)
#(EQ:#(A***::::::::::)#(DD:A***)::(
    #(GO:@2)):(
    #(GO:@3)))
```

#(PB:N:D1:D2) PRINT BINARY

The string N is displayed as a bit string on the screen, starting at bit D1, for D2 bits. Forms storage is unchanged, and there is no replacement value. If D1 is omitted, it is assumed to be 1. If D2 is omitted, it is assumed to be 255. If D2 exceeds 255 it is assumed to be 255. Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(DS:A***:#(EQ:@2::0:(#(SU:@2:1)))
#(PS:#(LOOP***:A***:
    #(CP:@3:256:(#(EQ:@3::256:@3)):256:@3):
    (#(TB:@1:#(A***):1:0))))
#(DD:A***)
```

#(SLIDE:X:MSS) DISPLAY SLIDE

The slide named X is displayed on the slide screen. If the slide is unavailable, a description appears on the CRT. If the wrong tray is mounted, a message to change it is sent. The slide is displayed for M minutes and SS seconds. If the last argument is blank, the slide is displayed for ten seconds. If there is a concurrent message, it is displayed at the same time as the slide. Forms storage is unchanged, and there is no replacement value.

Any previous definitions of A***, B***, T***, ID*** are deleted. FILE is expected to contain the name of the current file. When the command is finished, TRAY1*** has the name of the tray mounted on projector 1, and TRAY2*** has the name of the tray mounted on projector 2. PROJ*** has the number of the currently used projector. PR1*** has a count of recent usage of projector 1 and PR2*** has a count of recent usage of projector 2. The form is as follows:

```
#(DS:ID***:@1)#(DS:T***:@2)
#(SC1*)#(DD:T***:ID***:A***:B***)
```

SC1*

```
#(EQ:#(FO:#(FILE):#(ID***):VLBL):1:(#(SC2*)):(
      #(TB:IOERR:11::(
          #(PS:#(FO:#(FILE):#(ID***):DSCRPTN))))))
```

```
SC2*

            #(EQ:#(TRAY1***):#(FO:#(FILE):#(ID***):TRY:A***)#(A***):(
                    #(AN:PR1***:1)#(DS:PROJ***:1)#(SC3*)):(
                    #(EQ:#(TRAY2***):#(A***):(
                        #(AN:PR2***:1)#(DS:PROJ***:2)#(SC3*)):(#(MOUNT)))))
SC3*

            #(DD:A***)#(PS:>\SO#(PROJ***)/
                    #(EQ:#(T***)::010:(#T***)))/
                    #(FO:#(FILE):#(ID***):PSTNS9::)////*[
                    #(FO:#(FILE):#(ID***):CNCRRNTM)<
                    >\SO#(PROJ***)//////*<)
                    #(PS:PRESS THE RUN BUTTON NOW.)
MOUNT

            #(PS: MOUNT SLIDE TRAY #(A***) ON PROJECTOR
            #(CP:#(PR1***):#(PR2***):(#(DS:B***:1)):(
                 #(DS:B***:1)):(#(DS:B***:2))) #(B***))
            #(DS:PR1***:0)#(DS:PR2***:0)#(DS:PR#(B***)***:1)
            #(DS:TRAY#(B***)***:#(A***))#(SC3*)
```

#(TAKEN:D:STEPNO1:STEPNO2) TAKEN THE SCRIPT

If the inquirer has taken the script numbered D, control transfers to STEPNO1, otherwise control transfers to STEPNO2.

If STEPNO2 is omitted, the next step is taken. Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(FO:#(NICK):TAKEN:ETAKEN:A***:(#(DD:A***)))
#(BO:A***:STAKEN)
#(TB:A***:@1:(#(DD:A***)#(GO:@2)):(
        #(DD:A***)
        #(EQ:@3:::(#(GO:@3)))))
```

#(WD:N:Z) WORD

The value of the function is the next word list N. If no word remains, the value is Z. The form pointer is advanced to the next item on the list. Word delimiters include . ? , ; : ! and space.

Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(DS:A***:#(CS:@1:(#(GO:@2))))
#(SS:A***:,: :,:;:?:(:):!)

#(DS:A***:##(CS:A***))
##(IN:@1:#(A***))
#(CC:@1)#(A***)#(DD:A***)
```

APPENDIX E

MINORCA - Paper #2

Dorothy Swithenbank
Arnold Smith

MINORCA - PAPER #2

Introduction

This paper contains (preliminary) specifications for the MINORCA language. The specifications for the language have been drawn up with the following goals in mind. Some of the goals appear to be almost contradictory, and none of them is precisely defined, so the language comes at best only approximately close to meeting them. Comments are invited as to where the language falls considerably short of meeting the goals, or indeed, as to how the goals themselves should be reformulated.

(1) The language should be sufficiently powerful to handle easily the requirements of ISVD, i.e., scripts written in MINORCA should be able to control the entire system for student-machine interaction. In particular, MINORCA should contain many if not most of the capabilities of ELIZA and TRAC.

(2) The language should be relatively simple to use by people whose main interest is not programming. Scripts should be easy to write and easy to read. Simple procedures, in particular, should be simple to specify.

(3) Relationships among scripts should be easy to define and easy to visualize. Flow of control and selected information should be easily passed from one script to another.

(4) Implementation of a processor for the language should be a tractable job. Script execution should be efficient enough so that response time from the terminal is not unreasonable.

General

The basic unit of structure in MINORCA is the script. One script may control an entire conversation, or several scripts may share control. Communication among scripts is kept to a somewhat restricted and formal level, with the idea that different scripts may well be written by different people, normally dealing with different areas of a problem. Within a script there are no restrictions on such things as branching of control and symbol referencing.

Levels

Each script is assigned a level. The level indicates where the script fits into the hierarchy of the system. Levels are numbered sequentially from zero (although they are usually represented symbolically -- see below). Level zero is the highest -- there will be probably only one script at level zero which will be a control script. A script can transfer control directly only to a script whose level is the same as, or lower than, its own (i.e. has a level number equal to, or greater than, its own). There are two indirect ways of getting back up the line which will be described below.

Symbols

Many things in a script -- strings, lists, statement numbers, data -- will be referred to by name, rather than explicitly. These names will all have the same general form, and will be referred to in this description as stringnames or sometimes as symbols. A stringname (symbol) may consist of any number of any character except ( ), and blank, as long as there is at least one letter. Furthermore, the following words are forbidden as stringnames: IF, IS, OF, OR, AND, THEN, EQUALS.

Comments

A scriptwriter may insert a comment after the end of any statement, either on the same line as the statement or on a new line, by preceding the comment with a dollar sign. Each new line of commentary must begin with a new dollar sign. Otherwise, there are no restrictions on the text of the comment.

SPECIFICATION STATEMENTS

Script name. Each script is given a name, which must be unique within the system. Its length is limited to 30 characters, which may include anything but slash or dollar sign. It is specified in the first line of the script. The format is

```
     SCRIPT name
e.g. SCRIPT ROCKING HORSE 2
```

End of script. The last line of a script must have the following format

```
     ENDSCRIPT
```

Level specification. The second line of a script must specify the level of the script. Its format is

LEVEL number
or LEVEL stringname

e.g. LEVEL 4
LEVEL SUBCATEGORY

The second format (symbolic) will normally be used. Only authorized level names should be used, because the value of the stringname must be defined in the level zero control script. Symbolic levels are primarily a convenience while the system is being built, so that intermediate levels may be easily added without changing all scripts. Normally, the values assigned to symbolic level names will not change, and the level of a script will be determined by what type of script it is.

Statement Names, Entry Points, Back-Up Points, Punt Points

For convenience, statements may be labeled. Any stringname may label a statement. In the past scripts were envisioned as a collection of frames. Each frame was labeled

*xxx.xx

where x was any digit. All statements within frames were labeled

xxx.xx

Any scriptwriter, who wishes to use this means of organizing a script, may use statement labels of the form

*xxx.xx or xxx.xx

All statement labels must begin in column one. Other than statement labels no other character may appear until column six on a punched card. The only exception to the above rule is that a hyphen must appear in column one to indicate the continuation of a statement. Each statement must begin on a new line.

One option for the frame statement, which should be used sparingly, enables an outside script to enter this script at this frame (normally other scripts branch to the beginning of the script). This defines an external label which is treated by the system as if it were another script name. The name must therefore conform to the requirements for a script name, including the requirement that it be unique within the system. Ordinary statement labels must be unique only within the script. The format for the external label is

ENTRY POINT name

where "name" is defined as the external label.

Another option specifies a convenient point that a student may back up to if confused. The format for this statement is

BACKUP POINT

Each time a BACKUP POINT statement is encountered it is stored on an internal list until ten locations are on the list. When the eleventh location is stored, the first will automatically be removed. Should the student wish to go back he would be returned to the last backup point encountered. If he wished to go back further he would go to the second last backup point encountered. Should a scriptwriter not specify logical backup points an internal system routine will attempt to find a place to resume the dialogue.

At times a student's response may not have been anticipated by the scriptwriter. The scriptwriter may wish to insert routines at various script levels to handle these problems. The beginning of one of these routines should be indicated by a statement of the format

PUNT POINT

When the normal methods of responding to student input have been exhausted and the machine is instructed to

PUNT

it will see if there is a PUNT POINT in its current script, of, for example, level 2. If there is no PUNT POINT in that level 2 script, it would see if there be a PUNT POINT in the level one script from which it came. If there is not, the machine would return control to the generalized punt routine in level zero.

PUNT POINT's, ENTRY POINT's, and BACKUP POINT's may have statement labels. One point may also be, for example, both a backup point and an entry point. Since ENTRY POINT and BACKUP POINT are separate statements, they should be on consecutive, separate lines. However, blank lines may occur any place within the program.

STRING MANIPULATION

The basic form of a string is a sequence of characters enclosed in double quotes. Any character at all may appear between the pairs of quotes, except double quotes and (perhaps) slash, but including single quotes, for instance.

e.g. "HER PENSIVE AQUAMARINE EYES DIDN'T BLINK"

A String may also be represented by a stringname, in which case the stringname may be used in place of the literal string it represents. To name a string, the SET command is used. Its format is

```
      SET stringname TO string
```

```
e.g.  SET COLOR TO "AQUAMARINE"
      SET QUALITY TO "PENSIVE"
      SET QUANTITY TO "A THOUSAND AND ONE"
```

The stringname COLOR is said to have the value AQUAMARINE after this statement. Its value can be changed at any time by doing another SET.

A string may also be represented as a concatenation (placing side by side) of smaller strings, so that

```
      "HER" QUALITY COLOR "EYES"
```

could be the same string as the first example above.

Pure numbers are taken as literal strings even without the quotes around them. Thus both

```
      SET COUNT TO 4
and   SET COUNT TO "4"
```

have the same meaning.

Throughout these specifications, when the word "stringname" appears in a statement format, it means that in that position a single name may be used, with the restrictions specified above for stringnames. When the word "string" appears, any of the above forms of a string may be used, as well as any of the forms to be specified below under List Manipulation. In particular, the word "string" may stand for a single literal string, or a single stringname, or a sequence of literal strings and stringnames. Every stringname is a string; but the converse is not true.

Lists

A list is a string consisting of a sequence of smaller strings separated by a special character. Although it is stored just like an ordinary string, and has a name just like any other string, a list is normally used in a different way. It is used, in fact, as its name implied, to keep a list of a number of separate items, each of which is a string of some kind.

MINORCA provides functions to create and maintain lists. To create a list, or to add an item, the form is

```
ON stringname PUT string
ONBOTTOM OF stringname PUT string
```

If "stringname" is a previously defined list, these functions add the new item "string" to the top (beginning) or bottom (end) of the list designated. If no such list yet exists, either function defines a list with the name given, and the item "string" becomes its first item. If "string" is or includes the name of another list, that entire list is added to the specified list. Lists may also be specified explicitly by using the set command. Each item of the list, that is not itself a list, must be enclosed in double quotes,

e.g. `SET COLLEGES TO "HARVARD" "YALE" "BROWN"`

To remove an item from a list, MINORCA has the command

```
FROM stringname DELETE string
```

If string matches some item on the list specified by stringname, the first occurrence of the item on the list is deleted. If there is no match, nothing is done. If string is the name of a list, then all items common to both lists are removed from the list specified by stringname. If string refers literally to an item on the list it must be in double quotes. For example, if stringname is ANIMAL and string is DOG, all elements of the list DOG are deleted from the list ANIMAL. However, if stringname is ANIMAL and string is "DOG", the element DOG is removed from the list ANIMAL.

List Manipulation

There are a number of functions of lists which can be used to generate strings, some of which are themselves lists. Any of these functions can be used wherever a statement format specifies "string". (Among other things this implies that some of the functions can be used recursively.)

The simplest of these functions has the form

```
ITEM string OF stringname
```

In this case the "string" is normally either a number of a stringname whose value is a number, and stringname is the name of a list,

```
e.g. ITEM # OF COLLEGELIST
  or ITEM N OF EQUIPMENT
```

If the list specified by "stringname" does not have enough items to make this a meaningful function, the result is a null string, or an empty list, a string with no characters, no string at all. Occasionally a list may be implicitly paired.

```
e.g.  stringname      pair 1        pair 2        pair 3
      SALARIES     LAWYER,15000,DOCTOR,20000,SOLDIER,5000
```

To receive the partner of the first member of the pair use the command

```
      ITEM FOLLOWING string ON stringname
```

The third list-handling function has the format

```
      TOP OF stringname
```

Again "stringname" is a list. The value of this function is the first item on the list, but when called for, this function also removed the first item from the list. For example, if the following two statements were performed in sequence

```
      SET VALUE1 TO TOP OF ALIST
      SET VALUE2 TO TOP OF ALIST
```

the result would be that VALUE1 and VALUE2 would be different (probably), and ALIST would have two fewer items than it did originally. The function ITEM 1 OF gives the same value as TOP OF, but the side effects are different.

The last two functions which may be substituted for strings are

```
      EQUALITIES OF stringname, stringname
      DIFFERENCES OF stringname, stringname
```

In each case the two stringnames refer to different lists, and the result is a list. EQUALITIES gives all the elements common to both lists; DIFFERENCES produces a list with all items on the first list that are not on the second.

Any of the list functions may, if necessary, be used on normal strings.

Differentiating Stringnames From Their Values

Occasionally it will be useful to have a stringname whose immediate value is another stringname, or it may be necessary to have a list of stringnames rather than a list of the explicit strings which they represent. One problem, for example, might be in the statement

```
      (1)  GO TO COLLEGE
   or
      (2)  IF ITEM 4 OF COLLEGE IS EQUAL TO "HARVARD" GO TO LOCATION
           1 ELSE CONTINUE
```

In (1) one may indeed wish to go to location COLLEGE. However, one may also wish to transfer to the location whose name is the string stored at COLLEGE. In (2) one may want to know if item 4 of list, COLLEGE is equal to "HARVARD". One may, however, wish to know if the string whose name is the value stored at item 4 of COLLEGE is equal to HARVARD. There are two proposed ways of distinguishing a value of an item from its name. One method is to precede the ambiguous term with VALUE OF. The statements would look like

```
GO TO VALUE OF COLLEGE
GO TO (COLLEGE)
GO TO VALUE OF ITEM 4 OF COLLEGE
GO TO (ITEM 4 OF COLLEGE)
```

Would the scriptwriters express a preference as to which method they would like implemented?

OUTPUT and INPUT

In the current version of MINORCA there is only one output statement, in two forms:

```
      CRT string
and   CRT+ string
```

Both forms display the string on the console cathode ray tube. The first form erases anything that may have been there before, the second adds the string to whatever is already displayed, starting on the first free line. Special characters for format control may be included in the string. A number following CRT will be interpreted to mean how many seconds the display should remain on the screen. Otherwise the display would be removed either when the next CRT statement was encountered or when the screen was filled.

There is also only one input statement

```
KEYBOARD
```

This causes the computer to wait for the next response from the typewriter keyboard. Whatever is typed in is stored for analysis on a string called INPUT. When type-in is complete, the script resumes at the statement following the KEYBOARD statement.

Associated with the input and output statements are the two control statements

```
      RECORD
and   RECORDOFF
```

These are really system control functions and will probably be executed only by the control script. When record is on, all output and all input relating to a particular console, together with the script name, frame number and statement number of the statements that generated and received it, is kept on a file for later examination of printout.

Arithmetic

MINORCA currently has only the most primitive of arithmetic capabilities. As soon as a need for more involved statistics arises more arithmetic functions can be easily implemented. There are two statements, each in two forms.

```
INCREASE stringname
INCREASE stringname BY string

DECREASE stringname
DECREASE stringname BY string
```

In each case "stringname" is assumed to be the name of a string with a numeric value. "String" is also normally either a number of another stringname with a numeric value. If the second part of the statement is not included, the value to be added or subtracted is taken to be one.

Branching

The principal branch instruction is of the form

```
GOTO string
```

"String" can be the label of any location or a string which contains the label of a location. A stringname that represents a string of one of these forms, etc.

```
e.g.  GO TO COLLEGES
      GO TO VALUE OF COLLEGES
   or GO TO (COLLEGES)
```

To get to another script, the appropriate instruction is

```
CALL name
```

where "name" is the name of a script or one of its entry points.

When the script wants to get back control after an excursion into another script or even just into another area within its own boundaries, the CALL or GOTO command is preceded by

```
STORE LOCATION string
```

where "string" has the same form as for a GOTO statement. Then when the remote script or section of script executes the statement

RETURN

control will be returned to the original script at the location specified by the STORE LOCATION instruction. If the remote script in turn wants to call on a third script, it may execute a STORE LOCATION instruction which will cause its address to be stacked on top of a system list, and each RETURN statement will restore the list to its previous state.

Since all scripts except the level zero control script will be called into action by a higher level script, the way for a script to relinquish control, to end, step, etc., is to execute a RETURN. Control then ultimately returns, as it should, to the control script.

ANALYSIS

The IF statement is the basic decision-making statement of MINORCA. It has seven elementary forms, which are

```
IF string EQUALS string THEN statement ELSE statement
IF string IS EQUAL TO string THEN statement ELSE statement
IF string IS NOT EQUAL TO string THEN statement ELSE statement
IF string IS GREATER THAN string THEN statement ELSE statement
IF string IS LESS THAN string THEN statement ELSE statement
IF string IS EMPTY THEN statement ELSE statement
IF string IS NOT EMPTY THEN statement ELSE statement
```

The first three forms (first two identical in function) are general tests for string equality, but can also be used on numbers. The 4th and 5th forms expect the strings to be explicitly or implicitly numeric. The last two forms are designed to test lists. Any statement may follow the THEN and ELSE, including another IF statement (this is discouraged). If the condition is satisfied, the statement following the THEN is executed, and that is succeeded by the next numbered statement in the script, bypassing the ELSE clause, if there is one. The ELSE clause may be omitted; if it is present, its statement is executed if the condition is not satisfied. Control again proceeds to the next statement in the program. If the ELSE clause is omitted, control proceeds directly to the next statement of the script when the condition is not met.

More conditional phrases of the form

```
      string condition string
or    string condition
```

can be added between the IF and the THEN. The various condition phrases are then joined by AND, OR, AND IF, OR IF. Since this is all very much like English construction it is much easier to understand than to describe. The result is compound statements such as

```
IF ALIST IS EMPTY AND AGE IS LESS THAN 14 THEN SET CHANGES TO "LOW"
IF A EQUALS B OR C EQUALS D AND IF E IS NOT EMPTY THEN CONTINUE ELSE GOTO F
```

The second example illustrates the use of the CONTINUE statement. It may be used anywhere as an ordinary statement. However, it does nothing. It also illustrates the importance of the hierarchy of the connectives. These connectives combine the two adjacent conditions to form one. This causes no problem is all the connectives in an IF statement are the same, but when they are not, the order in which the conditions are combined affects the meaning of the statement. The connective hierarchy, in the order in which they are applied is AND, OR, AND IF, OR IF.

Keywords, Decomposition and Recomposition

For this discussion the reader is assumed to have some familiarity with the ELIZA language. The differences between ELIZA and MINORCA in the area of keywords, decomposition and recomposition will be discussed.

One of the main problems with ELIZA is the fact that most dictionaries are only of use to the small script in which they are specified. In MINORCA dictionaries may be specified anywhere and used anywhere. Each dictionary must begin with the statement

```
DICTIONARY stringname
```

and end with the statement

```
END DICTIONARY
```

In between these two statements any number of keywords may be defined in the form

```
      keyword priority number (decomposition rule 1) statement
                              (decomposition rule 2) statement
e.g.  College  100 (0 interested 0 College)  GO TO LOCATION.1
                   (0 not interested 0 College)  GO TO LOCATION.2
                   (0 College 0)  ON OUTPUT PUT ITEM 1 OF ANALYSIS, CONTINUE
```

The keyword is the word for which one is looking (Each keyword must be preceded by a minus sign. Blanks are ignored so keywords may be phrases.). The priority number (which if omitted is assumed to be 0) can give a keyword a higher value than the rest, thus causing it to be looked for first. The decomposition rule is matched. *If there is no match, the next decomposition rule is scanned, then the next keyword until the end of the dictionary is reached. If no match is encountered in the dictionary, control is returned to the next statement in the script from which control originated.

In MINORCA keyword analysis of a string is called for explicitly by the ANALYZE statement. Any string may be analyzed, although INPUT (which contains the latest typed response) is assumed if a string is not not specified. The form of the statement is

```
ANALYZE IN stringname
```

or

```
ANALYZE string IN  stringname
```

String is included, is the string to be analyzed. Stringname is the name of a dictionary of keywords.

One call for analysis might look like

```
SET LOCATION.1 TO "COLLEGE"
SET LOCATION.2 TO "JOB"
SET LOCATION.3 TO "MILITARY
ANALYZE IN DECISION
ANALYZE IN DON'T.KNOW
```

The first three statements establish transfer points from the dictionary. The input will first be analyzed in DICTIONARY DECISION. If no match is encountered in that dictionary it will then be looked up in DICTIONARY DON'T KNOW.

---

[*] Note that the list called ANALYSIS in the above dictionary refers to elements of the decomposition rule. Item 1 of analysis is all words preceeding the word college; item 2 is college.

APPENDIX

This appendix provides a formal definition of the MINORCA language. It is a definition of the form of the language rather than a grammar for generating the language, although the definition could be converted to a grammar without much difficulty.

The form used here is a variation on Backus normal. Single lowercase words are used for elements of the language being defined. Square brackets [ ] enclose an optional component of a right hand part of a rule. Square brackets followed by a degree sign [ ]° encloses a component which can occur any number of times or not at all. Exclamation point indicates a choice between the components it separates. The terminal chracter period is represented by the word "period" since it is also used as a metacharacter. Carriage return or end of card is represented by "end-of-line".

Blanks are a special problem. Outside a literal string, wherever one blank may or must occur, any number may be used. In certain cases, the required presence of at least one blank is indicated by the word "blanks". Otherwise, the general rule is that every stringname and number must be delimited by at least one blank if it is not otherwise delimited by a comma or a double quote, and that a statement label must begin at the beginning of a line.

```
char ::= A!B!C!...!Y!Z!0!1!2!...!9! period!=!+!-!'!*!/!$

digit ::= 0!1!2!...!9

nondigit ::= A!B!C!...!Y!Z! period!=!...!$

xchar ::= char! blank!,!(!)

number ::= [+!-] digit [digit]°

literal ::= "[xchar]°"!number

stringname ::= [char]° nondigit [char]°! VALUE OF string! (string)

label ::= stringname

string ::= string [string]°

string ::= literal !

           stringname !

           ITEM string OF string !

           TOP OF stringname !
```

```
             EQUALITIES OF stringname, stringname !
             DIFFERENCES OF stringname, stringname!
             ITEM FOLLOWING string ON stringname
scriptname ::= xchar [xchar]°
```

```
script ::=  SCRIPT scriptname end-of-line
            LEVEL string [comment] end-of-line
            [ [statement [comment]! comment] end-of-line]°
            ENDSCRIPT end-of-line
comment ::= *[xchar]°
statement ::= blanks COMMON stringname [,stringname]
          ::= label DICTIONARY end-of-line
                [dictionaryentry end-of-line]°
              blanks ENDICTIONARY
          ::= blanks ENTRY POINT scriptname
          ::= blanks BACKUP POINT
          ::= blanks PUNTENTRY
          ::= [label] blanks command
command   ::= CRT [+] [(number)] string
          ::= KEYBOARD
          ::= SET stringname TO string [,string]
          ::= ON stringname PUT string [,string]
          ::= FROM stringname DELETE string [,string]
          ::= INCREASE stringname [BY string]
          ::= DECREASE stringname [BY string]
          ::= ANALYZE[stringname,] stringname
```

APPENDIX F

The Addition of Statistical Primitives to TRAC

Charles S. Wetherell

```
          ::= CALL scriptname

          ::= RETURN

          ::= PUNT

          ::= BACKUP

          ::= IF condition [AND condition! OR condition! AND IF
              condition! OR IF condition]°

                     - THEN command [ELSE command]

condition ::= string EQUALS string

          ::= string IS EQUAL TO string

          ::= string IS NOT EQUAL TO string

          ::= string IS LESS THAN string

          ::= string IS GREATER THAN string

          ::= string IS EMPTY

          ::= string IS NOT EMPTY

dictionaryentry ::= EQUIVALENTS literal, literal [,literal]°

                ::= EQUIVALENTS literal=literal [=literal]:

                ::= literal decompositionrule end-of-line

                    [blanks command end-of-line]°

decompositionrule ::= (element [element]°)

element ::= number! literal! (stringname)
```

Within a dictionary entry some of the usual rules are changed slightly. An element that is a literal need not have the usual double quotes. Also the form (n), where n is a number, will be recognized as a valid string in the commands following a decomposition rule. (Its value is the match for the n*th* element in the decomposition rule.

INTRODUCTION

This paper presents a package of statistical operators designed to operate within the context of the string handling language TRAC.[*] The need for this package arose within a research project conducted by the Harvard Graduate School of Education. Before the work was very far advanced, it was clear that a careful analysis of the structure of TRAC and of the statistical problems to be solved was needed. Once this analysis was made, the implementation of the package was relatively easy. However, several limitations were recognized and when the work was done, it was clear that the package could be extended in several directions. I shall discuss the history, design, and future of the package and the use of these extensions in other TRAC systems.

---

[*] TRAC is the trademark of Rockford Research Institute, Inc., Cambridge, Massachusetts, for its string handling language.

THE BACKGROUND OF THE PROBLEM

The need for a statistical package within the language TRAC first arose at the Information System for Vocational Decisions Project (ISVD), funded by the United States Office of Education and administered by the Harvard Graduate School of Education (Tiedeman, 1965). This project, a large one, will develop methods to train students in the methods of decision-making, particularly in the choice of vocations, through the use of computer-controlled consoles and individualized personal guidance. Students will have access, *via* the computer, to large data bases of information about decision-making, the "world of work," the local employment and economic situation, and their own personal data (perhaps compounded with data about other students). The project investigators will have to collect, collate, evaluate, and compile this data. To do this, they must have a statistical picture of the student's activities with the data base and statistical methods to transform the raw data into a form suitable to students.

The student's interaction with the computer will be controlled by a "script". As the student "plays" the script, he will, possibly unknowingly, call various programs to life and cause informations to be retrieved from the several data bases. This information, however, may not appropriate for the student as it stands. If it is not, the script will cause the proper statistical actions to be taken. These may be as simple as scaling the numbers and as complicated as discriminant analysis. It is also the responsibility of the script to interpret the information for the student within the context of the particular play of the script in which the student is engaged.

This all assumes a control language with which the investigator can manipulate the programs and data structures of the system into a script. Basically, that control language is TRAC. Actually, the language which script writers use will be considerably more sophisticated. Presently, this higher language is interpreted by the TRAC processor and certain TRAC procedures at the time of the execution. Eventually, it may be compiled at the time it is written, with few references to the TRAC interpreter.

However, the compiler may be written in TRAC and the TRAC interpreter may have full control of the computer while the compiled program is run. The object code may also be TRAC.

Since the investigator needs statistical powers in the two contexts and since both contexts will involve the use of TRAC, it is clearly necessary to attempt the statistical work within TRAC. The solution is to add a block of primitives to TRAC which will perform the operations needed without disturbing the present set of primitives. Fortunately, there exist a set of statistical operators which fit the form of TRAC primitives and which produce the statistics which ISVD needs.

THE STRUCTURE OF TRAC

TRAC is a string handling language proposed and first presented by Mooers and Deutsch (1965 and 1966). It bears a good deal of resemblance to several other string and list processing languages, most notably LISP, GPM, COMIT, and SNOBOL. The basic entities with which it operates are strings of BCD characters and it manipulates these with a relatively simple interpreter. The basic philosophy of TRAC is that of a macro-expansion language. More on TRAC will be found in Appendix A, Mooers' definitive article (1966).

The arithmetic capability of TRAC is very limited. The arithmetic operations all perform Arabic arithmetic on BCD integer strings, that is, they operate right to left on strings of indefinite length by manipulation of BCD characters. While this provides arithmetic of a natural sort, and may be expanded to simulate all arithmetic operations, it is inefficient for anything but simple manipulation of integers. Thus, if a statistical package is to be added to TRAC, a new set of arithmetic operators will be needed which will take advantage of the floating point arithmetic provided by most computers. However, they must have the form of the TRAC primitives and the data on which they operate will have to conform to the data storage procedures of TRAC.

The Beaton set of linear operators is precisely this type of package. These operators map various combinations of matrices, vectors, and scalars onto other combinations. With the operators, all of lineary parametric statistics* may be done, that is, they are a complete basis for most of the standard statistics. Further, each operator may be cast in the form of a function with a fixed number of arguments. Each argument is itself an array of real numbers. With these operators and supporting functions to manipulate arrays, a statistical package for ISVD could be constructed.

---

*"Statistics" will be used in two senses in this paper. In one sense, it refers to a body of knowledge lying within mathematics and in the other, it refers to individual numbers containing information about certain systems. The usage should be clear from context.

TRAC, as designed, is modular. That is, the addition of a primitive function involves only the addition of one more name to the list of functions available to the processor. It would be possible to have an interpreter which was impotent because it was not supplied with primitives. Any given function may be eliminated and the others will not be affected. No function relies on any other and all common pieces of code are contained within the body of the interpreter. The addition of this statistical package can be made without affecting the rest of TRAC. ISVD, and other users, will be able to add or delete the package at will. I shall return to this point later.

THE BEATON OPERATORS

The basic concept behind the package of functions which I shall develop for inclusion in TRAC is that advanced by Beaton: simply, every statistical operation on a linear model may be handled by a "small" set of operators. In any given situation which one wishes to study statistically, there will be several variables to be measured, and of these variables, it is to be hoped that some will depend on the others. A model is assumed for the dependency and various statistics may be calculated to check the validity of the model. Beaton showed that if the observations were arranged in a matrix in a certain way, then there is a set of about ten operators that will calculate, singly and in composition, every statistic based on a linear model for the dependency relation. Further, it can be shown that any curvilinear model may be reduced to a linear model. Thus, the operators of Beaton suffice to do almost all possible statistical calculation. They fail only when the statistic needed is one based on frequency counts or similar discrete groupings of continuous variables. On the other hand, there are, compared to the number of linear statistics, few of these mavericks, albeit several of them are quite important.

The Beaton operators take as arguments arrays; scalars, vectors, and matrices. Data in TRAC, BCD strings, are stored as sections of a long vector. It has long been known that an array may be stored as a vector by developing an appropriate mapping between the subscripts of array elements and the linear subscript on the vector. Thus, the statistical data with which ISVD will transact may be handled by TRAC without doing violence to the storage mechanism of TRAC. The set of Beaton operators and the attendant array manipulation primitives fit the specification which we laid out for the statistical package. Following this reasoning, ISVD decided to implement the package within TRAC.

THE STORAGE OF ARRAYS

When the design of this package was begun, I realized that limiting arrays to the two dimensions required by the Beaton operators was an artificial restriction. So arrays may have arbitrarily high dimensions: that is, the dimension may be any integer from zero on up. Similarly, subscripts may take any negative, zero, or positive integral value. The typical array will be n-dimensional and the subscripts along dimension i will range from a lower bound l_i to an upper bound u_i, with the only restrictions being that l_i and u_i both be integers and that l_i is less than or equal to u_i. The difference $u_i - l_i$ will be known as the range of subscripts along the dimension i. Arrays elements may be any real numbers, as expressed in decimal notation.

Of course, no computer is going to be able to handle an array of dimension 40 and script range 10 along each dimension. That would require at least 10^{40} cells of storage and no computer comes near that figure. An array with an upper subscript bound of 123×10^{67} would also be difficult, even if the lower bound along that dimension were the same; similarly an array with an element of the size 10^{80} would be difficult. The reason for these problems is that we would like to make our array storage and arithmetic use the arithmetic capacities of the machines upon which the primitives are run. But most machines have a limited range of sizes of numbers which they can accept and manipulate. They also all have severely limited storage capacities. So, although the coding for the package is in ALGOL without reference to a specific machine or to the inherent hardware limitations of most computers, these bounds will have to be kept in mind in any actual implementation.

The storage of strings in TRAC makes some concession to the structure of computers. Intuitively, a string is a sequence of characters, linearly ordered and finite, possibly having a name. Mooers' method of storage makes this concept somewhat more explicit so that it may be mechanized. In the process, some conceptual simplicity is lost. The concession formalizes the relation of string and name and required the use of certain heading information which travels with the string. The amalgam of string, name, and heading is a "form".

The heading information explicitly points to the ends of the string and the name and from the structure of a form, pointers to their beginnings may be calculated. It also contains some abstruse material of use to the interpreter. The actual order of these elements is

<type> <hash-code> <internal-text-pointer> <end-of-text-pointer>
<end-of-name-pointer> <text-string> <name-string>

Each element, except the <text-string> and the <name-string> are the BCD strings which were to be stored. The <type> is a code which designates the type of the string; presently there are primitive, string, and machine type codes and I am adding array type. The <hash-code> is a technical device used by the interpreter. It is described for completeness' sake only. The <internal-text-pointer> is an index manipulated by several of the original primitives. It will not be used by any of the primitives in the array package and will be employed for a different purpose. The <end-of-text-pointer> contains the displacement of the last cell of text from the <type> and the <end-of-name-pointer> contains the displacement of the last character of the name string. These last two pointers simply formalize the concept of the ends of the string and name.

If the TRAC interpreter is given the command

```
#(ds,waldo,This is a string.)
```

it will create a form which looks like

```
Cell 00  <type> = 2
Cell 01  <hash-code>                     }   We use neither
Cell 02  <internal-text-pointer>             of these.
Cell 03  <end-of-text-pointer> = 21
Cell 04  <end-of-name-pointer> = 26
Cell 05   'T' (i.e., the literal BCD character T)
Cell 06   'h'
Cell 07   'i'
Cell 08   's'
Cell 09   ' '
Cell 10   'i'
Cell 11   's'
Cell 12   ' '
Cell 13   'a'
Cell 14   ' '
```

Cell 15	's'
Cell 16	't'
Cell 17	'r'
Cell 18	'i'
Cell 19	'n'
Cell 20	'g'
Cell 21	'.'
Cell 22	'w'
Cell 23	'a'
Cell 24	'l'
Cell 25	'd'
Cell 26	'o'

The form will be positioned in a long integer typed vector in storage, the "f" vector, with only the address of the <type> of the first form kept as a reference point. Forms are found when needed by a leap-frogging technique, using the <end-of-name> pointer to find the next form when the form being inspected does not fit specifications of the one being searched for. The forms are laid in storage at the right end of the "f" vector, with new forms added at the left. Thus storage looks like

free-storage:form-n...:form-3:form-2:form-1

with low numbers denoting older forms. The whole mechanism is relatively simple and makes it easy to houseclean and garbage collect storage. More on this is in Mooers, 1966.

The problem is to fit array storage in with this well-established scheme. A method to map arrays onto vectors is well-known and I shall use the one described in the MAD manual. Combining it with TRAC will create an efficient and flexible storage method, at the same time preserving the structure and mechanisms of TRAC.

STORAGE DESIGN

The <type> for arrays will be 4. The <hash-code> will be left as it is now. The <internal-text-pointer> will not be needed for the package and so is renamed the <dimension>. This will be an integer from zero up and will be the dimension of the array which the form stores. The next two elements of the form will remain as they were, as will the <name-string>. The major change comes in the <text-string>, which now will contain a combination of integers and real numbers. The integers will be subscripting information and the real numbers the actual elements of the array.

To convert the subscripts naming a particular element of the array into a position in this vector, a mapping function must be used. If the matrix is a scalar, there is no mapping function and the value of the scalar is simply contained in one word, the fifth word of the <type> and the <end-of-text-pointer> contains the integer 5. If the array has dimension d greater than zero and if l_i is the lower subscript bound along dimension i and u_i the upper, where i is between 1 and d, then we can define a number b by

$$1)\ b = 6 + 2d - \sum_{i=1}^{d} \{ \prod_{j=i}^{d-1} (u_j - l_j + 1) \} (l_{d-i+1} - 1)$$

The number b will be known as the "base" of the array mapping function. Now is an element $a_{s_d s_{d-1} \ldots s_2 s_1}$ of the array is given and we wish to calculate the linear subscript of the element, then

$$2)\ r = \sum_{i=1}^{d} \{ \prod_{j=1}^{d-i} (u_j - l_j) \} (s_{d-i+1} - 1) + b$$

The <type> cell of the form is (arbitrarily) assigned the linear subscript zero. This means that the array element with lowest possible subscripts will have linear subscript 6 + 2d. This leaves precisely enough room for the subscript information, required for the calculation of these linear subscripts, between the <end-of-name-pointer> and the first array element.

The format of the text string will be

$$\langle base\rangle\ \langle l_1\rangle\ \langle u_1\rangle\ \langle l_2\rangle\ \ldots\ \langle l_d\rangle\ \langle u_d\rangle$$

$$\langle a_{l_d l_{d-1} \ldots l_1}\rangle\ \ldots\ \langle a_{l_d l_{d-1} \ldots u_1}\rangle$$

$$\langle a_{l_d l_{d-1} \ldots l_2 + 1\ l_1}\rangle\ \ldots$$

$$\vdots$$

$$\ldots\qquad\qquad \langle a_{u_d u_{d-1} \ldots u_1}\rangle$$

The elements of the array are laid into the form with their right-most subscripts varying most rapidly. Also, the right-most subscript bounds, l_1 and u_1, occur on the left end of the form. This makes access to them easier in later computations.

The ALGOL array "waldo", declared

array waldo[-1:0,3:5,-2:0]

has a total size of 18 elements and a dimension of 3. The subscript information for "waldo" is

$l_3 = -1;\ \ u_3 = 0;\ \ range_3 = 1$

$l_2 = 3;\ \ u_2 = 5;\ \ range_2 = 2$

$l_1 = -2;\ \ u_1 = 0;\ \ range_1 = 2.$

3) $b = 6 + 2\cdot 3 - [3\cdot 3\cdot(-1 - 1) + 3\cdot(3 - 1) + (-2 - 1)]$
$= 12 - [-18 + 6 - 3]$
$= 27$

The element $waldo_{-1,3,-2}$ maps to the linear element

4) $r = [3\cdot 3\ (-1 - 1) + 3\cdot(3-1) + (-2 - 1)] + 27$
$= -18 + 6 - 3 + 27$
$= 12$

This is the first element of the array in the form and is precisely 6+2d words from the <type> cell. Similarly, $waldo_{0,5,0}$ has linear subscript

5) $r = [3 \cdot 3\ (0 - 1) + 3\ (5 - 1) + (0 - 1) + 27$
$= -9 + 12 - 1 + 27$
$= 29$

which is 17 cells from the first element and which uses the 18th word of array storage. This is precisely correct for an array of 18 elements. This example array will look like the following diagram when it is stored as a form:

Cell 00	<type> = 4 (for array type)
Cell 01	<hash-code>
Cell 02	<dimension> = 3
Cell 03	<end-of-text-pointer> = 29
Cell 04	<end-of-name-pointer> = 34
Cell 05	<base> = 27
Cell 06	<l_1> = -2
Cell 07	<u_1> = 0
Cell 08	<l_2> = 3
Cell 09	<u_2> = 5
Cell 10	<l_3> = -1
Cell 11	<u_3> = 0
Cell 12	$\text{waldo}_{-1,3,-2}$
Cell 13	$\text{waldo}_{-1,3,-1}$
Cell 14	$\text{waldo}_{-1,3,0}$
Cell 15	$\text{waldo}_{-1,4,-2}$
...	
Cell 29	$\text{waldo}_{0,5,0}$
Cell 30	'w' (i.e., the literal BCD character w)
Cell 31	'a'
Cell 32	'l'
Cell 33	'd'
Cell 34	'o'

There is one difficulty with the implementation of this plan. In the "f" vector there will be integers, real numbers, and BCD characters mixed together. Fortunately, the BCD characters may be coded as integers, but there is no way to mix real and integer quantities in the same vector in ALGOL. One solution is to add a vector for the storage of real quantities which will be controlled by the same stack pointers as the "f" vector. If this is done, however, there will be great gaps in both the integer and real storage, alternating with one another. Wherever a real number has been stored, there will be a blank space in the "f" vector and *vice versa*. A solution which avoids this problem is storing pointers to the real vector with the heading information for the form. The <text-string> for the form would then be packed down into real vector and eliminated from the "f" vector. However, arrays handled this way would be difficult to manipulate. With both these solutions, a fifth vector is added to TRAC's storage system, and this is also unfortunate, for the four vector storage system of TRAC makes it easy to do variable storage allocation. This possibility would be destroyed by the addition of a fifth vector.

The approach I have taken is to add the fifth vector, the "a" vector, using the stack controls for the "f" vector. However, in the actual implementation of the package, I will declare the two vectors to be "equivalent", under the MAD or FORTRAN meaning of the word. This will mean that there are really only four vectors and that the program can fetch and store in either real or integer mode from any element of the combined "a-f" vector. In any implementation, it would even be worthwhile to sabotage the compiled object code to achieve this result, even if the language in which the package is written at source level does not allow the declaration of equivalence, as ALGOL does not.

THE PRIMITIVES

In this section the primitives are described and their definitions set out.

Array Definition

$\#(as,N,d,l_d,u_d,l_{d-1},\ldots,l_1,u_1)$. 2d + 3 arguments.

The array named N is defined with dimension d and subscript bound pairs l_i and u_i for i between 1 and d. N is a BCD string and d and the bounds pairs are BCD integers. Extraneous arguments are ignored. If there are fewer than d bounds pairs, the function is not executed. The elements of the array are all set to zero. This primitive corresponds, in some sense, to the ALGOL

array $N[l_d:u_d,l_{d-1},\ldots,l_1:u_1]$;

As with all forms storage, any form of similar name is erased from storage, whatever its type.

Dimension Value

$\#(av,N)$. 2 arguments.

The dimension of the array named N is returned as a BCD integer. If N is not an array or if N does not exist, the value is null (not zero).

Subscript Bound Values

$\#(ab,N,i,s)$. 4 arguments.

The value of the sth (i.e., upper or lower, coded 1 or 0, respectively) subscript bound of the array N along dimension i is returned as a signed BCD integer (the sign is used only if the value is negative). If i is greater than the dimension of N or if N is not an array or does not exist, or if s is not zero or one, the value is null.

Convert to Integer

$\#(ac,N,s_d,s_{d-1},\ldots,s_1)$. 2 + d arguments.

This primitive returns as value the greatest integer less than or equal to $N_{s_d,s_{d-1},\ldots,s_1}$, where d is the dimension of N. If there are not enough subscripts specified, if a subscript is out of bounds, if N is not an array, or if N does not exist, the value of the function is null. The returned value is a signed BCD integer.

Set Array Value

#(ae,N,s_d,s_{d-1},...,s_1,M,t_f,t_{f-1},...,t_1). 3 + d + f arguments.

This primitive replaces the value of $N_{s_d,s_{d-1},\ldots,s_1}$ with the value of $M_{t_f,t_{f-1},\ldots,t_1}$, where d and f are the dimensions of N and M, respectively, and the s_i and the t_j are BCD integers. If either M or N does not exist, or either M or N is not an array, or if any subscript is out of bounds, or if there are not enough subscripts, the replacement is not performed. The function has null value.

Read Array

#(ra,element-1,element-2,...,element-n). At least one argument.

This is simplified I/O for arrays. Input is one real value per line on the console. Each "element-i" is either an array name, or an array name followed by an asterisk, subscripts, and another asterisk. The subscripts are BCD integers and are separated by asterisks. An array which does not exist, an element which is not an array, or one for which there are not enough subscripts or for which the subscripts are out of bounds will be ignored. If an element is simply an array name, the entire array is read in, the rightmost subscript varying most rapidly. Excess subscripts within an element will be ignored. The value of the function is null. The numbers read from the console will be in "F" or "E" notation as that is understood within MAD. Different implementations may give different error indications depending on local I/O routines or different forms for the input.

Print Array

#(pa,element-1,element-2,...,element-n). At least one argument.

This is similar to "ra", except that arrays or array elements will be printed on the console in "E" notation and a simplified format.

Read by Format

```
#(rf,format,device,element-1,element-2,...,element-n).
```

At least three arguments.

The format string is of the same nature as that used in FORTRAN or MAD. The "device" is a number or name which the local operating system may recognize as a call for use of an input device. The input is read off the device named according to the format string. The format string is an ordinary BCD string in TRAC which also conforms to the local rules for format statements. Error indication and recovery will depend on the local system. As in "ra", elements in error will be ignored. The value is null.

Print by Format

```
#(pf,format,device,element-1,element-2,...,element-n).
```

At least three arguments.

The same as "rf", except that the values are read from the elements named onto the device.

Array Addition

`#(aa,A,B,C,Z).` Five arguments.

If A, B, and C are three arrays of the same dimension and of the same range along each dimension, then C is replaced by the element sum of A and B. If A is a scalar and B and C are similar[*] arrays, then C is replaced by B with A added to each element. The value of the function is either null or Z. The value is Z only when the addition is unsuccessful, i.e., when one of A, B, or C is not an array or does not exist, or when there is a dimension or subscript range error. Z is any BCD string and is placed in the active or neutral string as the call for the function was active or neutral.[**]

---

[*] Two arrays are similar if they are of the same dimension, and if the subscript range along each dimension is the same. The arrays need not have the same subscript bounds pairs; it is only necessary that $u_i - l_i = u_i' - l_i'$ for each dimension.

[**] This set of error conditions, i.e., wrong dimension, subscript out of bounds, non-existence of an array, or form not of array type will be known as the standard errors and the placement of Z in the appropriate setting will be known as a "branch to Z" or a "z-branch".

Array Multiplication

#(am,A,B,C,Z). Five arguments

The arrays A and B are multiplied together in the manner normal to methematics and the result is placed in C. If A and B are scalars or are vectors of the same length, C must be s scalar. A and B must always be of the same dimension; the standard identification of a vector with a column matrix will not be allowed, for example. The one exception is that if A is a scalar and B and C are similar matrices, C is the element by element multiple of B by A. In any case, C needs only the right subscript ranges and dimension to be a resultant matrix; the actual size of the subscripts is not checked. The value is null. A branch to Z is taken on the standard errors.

Array Inversion

#(ai,A,B,D,Z). Five arguments.

If A and B are similar square matrices and D is a scalar, then the inverse of A is placed in B and D contains the determinant of A, if the matrix A is non-singular. If the matrix A is singular, both B and D are set to zero and the branch to Z is taken, as it is on the standard errors. The inverse will be calculated by a combined direct and iterative method and will be considered correct when it falls within a pre-assigned tolerance.

Array Transposition

#(at,A,B,Z). Four arguments.

If A and B are matrices and if the subscript ranges of B are the reverse of those of A, then B is replaced by the transpose of A. The standard errors cause a branch to Z.

Beaton Routine DMD

#(af,A,B,C,Z). Five arguments.

If A and C are similar square matrices and B is a vector with the same length, then

$$C_{ij} = A_{ij}/(B_i \cdot B_j)$$

where i and j are normalized to lie between zero and the subscript range, inclusive.[*] If A, B, and C are all scalars, then

$$C = A/B^2.$$

If any of the B_i are zero or if any of the standard errors occur, the function is not performed and the branch to Z is taken. The value of the function is null.

Beaton Routine SCP

#(ag,A,B,C,Z). Five arguments.

If A and C are similar square matrices and B is a vector of the same length, then

$$C_{ij} = A_{ij} \cdot B_i \cdot B_j$$

If A, B, and C are scalars, then

$$C = AB^2$$

The standard errors cause a branch to Z. The value is null.

Beaton Routine SWP

#(ah,A,B,k,Z). Five arguments.

If A and B are similar square matrices with subscript range n and k is an integer with

$$0 \leq k \leq n$$

and $A_{kk} \neq 0$, then

$$B_{ij} = A_{ij} - \frac{A_{ik} - A_{kj}}{A_{kk}} \quad , \quad \begin{array}{l} i,j = 0,\ldots,n \\ i,j \neq k \end{array}$$

$$B_{kj} = A_{kj}/A_{kk}, \quad j \neq k$$

---

[*] In all these functions, we shall think of the subscripts as lying between zero and the range for the particular subscript. This is, of course, not true, since subscripts may lie between any two arbitrary bounds. However, thinking this way makes the mathematics clearer and to be correct, one need only add the appropriate lower subscript bound to each subscript.

$$B_{ik} = A_{ik}/A_{kk}, \quad i \neq k$$

$$B_{kk} = 1/A_{kk}.$$

If $A_{kk} = 0$ or one of the standard errors occurs, the branch to Z is taken. The value is null.

Beaton Routine DVEC

#(aj,A,B,Z). Four arguments.

If A is a square matrix and B is a vector of the same length and n is range of the subscripts, then if

$$A_{ii} \geq 0 \text{ for } i = 0,\ldots,n$$

then $$B_i = \sqrt{A_{ii}}$$

If any element of the diagonal of A is less than zero or if any of the standard errors occur, the branch to Z is taken. The value is null.

Beaton Routine DIRPRD

#(ak,A,B,C,Z). Five arguments.

If A is a vector with subscript range m and V is a vector with subscript range n and C is a vector with subscript range (n+1) (m+1)-1, then

$$C_{mi + j} = A_i \cdot B_j, \quad i = 0,\ldots,m; \quad j = 0,\ldots,n.$$

If a standard error occurs, the branch to Z is taken. Otherwise the value is null.

Scalar Absolute Value

(sa,A,B,Z). Four arguments.

If A and B are scalars, B is replaced by the absolute value of A. On the standard errors a branch to Z is taken. The value is null.

Scalar Exponential

#(se,A,B,Z). Four arguments.

If A and B are scalars, B is replaced with e^A. On the standard errors, the branch to Z is taken. The value is null.

Scalar Logarithm

#(sl,A,B,Z). Four arguments.

If A and B are scalars and A is greater than zero, then B is replaced by the natural logarithm of A. If A is less than or equal to zero or a standard error occurs, the branch to zero is taken. The value is null.

Scalar Power

#(sp,A,B,C,Z). Five arguments.

If A, B, and C are all scalars, then C is replaced by A^B as long as that operation results in a well-defined real number. If it does not, or if there is a standard error, the branch to Z is taken. The value is null.

THE CODING AND IMPLEMENTATION

This set of TRAC operators has been built on the basis of the standard TRAC interpreter as described by Mooers (1966) and as coded in ALGOL by Mooers and Deutsch (1966). This coding has been used as the model for a TRAC interpreter in MAD, to be run on the MIT-CTSS time-sharing system. The MAD-TRAC system has been running for about eight months at ISVD. I have coded the primitives in ALGOL to match that of Mooers and Deutsch for two reasons: the coding will be easily usable in other TRAC systems based on the ALGOL-TRAC and it will be easily translatable for use with the ISVD interpreter. The coding presented here is the ALGOL, but it is only the first step in a two-step project. The second one will be taken as needed within ISVD.

The only additions I have made to ALGOL-TRAC are the variables "charmi" and "charpl", containing the BCD characters "-" and "+" respectively, the switch diag, which is used for diagnostics and error recovery, and several other procedures described below. Each of the primitives is coded as a block which will fit with the other primitives in the interpreter without mutual interference. These blocks may only be entered at the top and all exits are unconditional transfers. Primitives freely manipulate the several pointers available in the interpreter and the elements of the various stacks, as do the procedures. The primitives are individually commented upon in the code. There has been a conscious effort to make the code clear; at times this effort has been at the cost of efficiency. In most cases the inefficiency is easy to correct. However, as the coding is intended as a guide to implementation and not as a strict set of instructions to programmers, I leave this to the taste of the reader and user.

Several primitives have not been coded. These are the ones concerning input and output of arrays. I feel that these will not be well done in ALGOL and that the local environment will be the most important determinant of their format. At ISVD, these routines will be written using MAD and the local system provided I/O routines. In other systems, it may well be necessary to use assembly language routines to code these primitives. Again, the user will have to decide on his own methods to implement these primitives.

Any method is legitimate as long as it conforms to the specification of the primitives.

One Beaton operator, SDG, has not been implemented as a primitive. At the present time, it does not seem necessary to use this routine. On the other hand, it is not a very difficult routine to code and may be added at any time. There are several other minor routines which are not central to Beaton's thesis, but which he includes. They may eventually be included in the ISVD system merely for the sake of completeness.

The procedures I have added to ALGOL-TRAC are mainly housekeeping and array indexing routines. In several places in the present code, a type of 4 will cause an error branch. It has been necessary to side-step this difficulty. This is primarily the reason for "getfa", "delete", and "zbranch". The routines "index" and "map" are used to calculate linear subscripts of elements of arrays. Finally, "convert", "expand", and "setup" handle the conversion between BCD integers in arguments and machine integers and *vice versa*. The routine "delete" is the only one which actually replaces a present routine. It was necessary to garbage clean the "a" vector.

As noted above, there is a tacit assumption in ALGOL that all arithmetic is of unlimited range. This, of course, is not true on any real machine. ALGOL also does not provide error recovery procedures for machine conditions like overflow and underflow. It is for the user to employ whatever mechanisms his machine provides to make the assumptions about arithmetic as true as possible and to effectively handle errors of various sorts. Guides beyond this gentle warning in this area would be futile.

THE CODED PRIMITIVES

The coding is in publication ALGOL, with the exception that the Boolean connectives '<u>and</u>', '<u>or</u>', and '<u>not</u>' are written out in full. As such, it has never been run on a machine. However, translations of portions of it in other languages have been attempted and successful. The code may have errors in detail, but I believe that overall structure of the routines is correct.

```
integer procedure convert(front,back);
  integer front,back;
  value front,back;
     begin
     integer i,j,hold;
     hold := 0;
     j := 1;
     if front = back
     then begin
          convert := 0;
          return;
          comment  In this case, the integer to be converted is
                   either the null string or is an argument from the
                   neutral string of no length (i.e., nonexistent);
          end
     else if w[front + 1] = charmi
             or w[front + 1] = charpl
          then begin
               j := (if w[front + 1] = charmi then -1 else 1);
               front := front + 1;
               end  This is the case when the integer in the neutral
                    string is preceded by a '+' or a '-' sign;
     for i := front +1 step 1 until back
     do   hold := hold × 10 + w[i];
     convert := j × hold;
     end  This procedure converts integers in the neutral string into
          their BCD representations.  The had of the BCD string is
          pointed to by 'front' and the tail by 'back'.  The value
          returned by convert is the machine integer;


procedure delete(i);
  integer i;
  value i;
     begin
```

```
        integer j,k;
        k := 1 + f[i + 4];
        for j := i - 1 step -1 until fn + 1
        do begin
           f[j + k] := f[j];
           f[j + k] := a[j];
           end;
        fn := fn + k;
        end  See 'delete' in Mooers' version of the interpreter;


integer procedure expand(pointer);
  integer pointer;
  value pointer;
        begin
        integer place,n,q,r;
        place := wl;
        n := abs(f[pointer]);
loop:   if place = wn
        then goto dia[2];
        if n < 10
        then begin
             w[place] = n;
             goto done;
             end  This part of the procedure does the expansion
                  when the machine word will expand into a single
                  integer;
        q := n/10;
        r := n - q × 10;
        w[place] := r;
        place := place - 1;
        n := q;
        goto loop;
done:   if sgn(f[pointer]) = -1
        then begin
             place := place - 1;
             if place = wn
             then goto dia[2];
             w[place] := charmi;
             end  If the machine integer was a negative number, the
                  BCD representation is preceded by a '-' sign;
        expand := place;
        end  This procedure produces the BCD representation of an in-
             teger from the machine representation.  Roughly, it is the
             inverse of convert.  The value of the function is a pointer
             to the leftmost character of the BCD integer in the neutral
             string.  The integer runs from this pointer to the end of
             the neutral string;
```

```
funaa: begin
       integer i;
       real temp;
       getfa(2,5);
       nq := rf;
       getfa(3,5);
       nq := rf;
       getfa(4,5);
       if f[np + 2] = 0 and f[nq + 20 = f[rf + 2]
       then begin
            for i : 1 step 1 until f[nq + 2]
            do   if f[nq + 5 + 2 × i] - f[nq + 4 + 2 × i]
                   ≠ f[rf + 5 + 2 × i] - f[rf + 4 + 2 × i]
                 then zbranch(5);
            temp := a[np + 5];
            for i := (if f[nq + 2] = 0 then 5 else 6 + 2 × f[nq + ])
             step 1 until f[nq + 3]
            do    a[rf + 1] := a[nq + 1] × temp;
            end  This portion adds the constant named by the first
                 argument (which was of zero dimension) to each element
                 of the second argument and stores the result in the
                 third argument.
       else if f[nq + 2] = f[np + 2] and f[nq + 2] = f[rf + 2]
            then begin
                 for i := 1 step 1 until f[nq + 2]
                 do   if f[np + 5 + 2 × i] = f[np + 4 + 2 × i]
                         ≠ f[nq + 5 + 2 × i] - f[nq + 4 + 2 × i]
                         or f[np + 5 + 2 × i] - f[np + 4 + 2 × i]
                         ≠ f[rf + 5 + 2 × i] - f[rf + 4 + 2 × i]
                      then zbranch(5);
                 for i := 6 + 2 × f[np + 2] step 1 until f[np + 3]
                 do a[rf + i] := a[np + i] + a[nq + i]
                 end  This portion checks to see if all the arguments
                      are of the same dimension and each dimension has
                      the same range (although not necessarily the same
                      bounds) and if this is so, adds the first array
                      to the second, element by element, and stores the
                      result in the third array.  This is an option
                      which will not occur if the first half of the
                      program has been executed;
       goto nullret;
       end;
```

```
funab:  begin
        integer sub,uorl;
        getfa(2,0);
        get(3);
        sub := convert(rp,rq);
        if sub > f[rf + 2] or sub < 1
        then goto nullret;
        get(4);
        uorl := convert(rp,rq);
        if not (uorl = 0 or uorl = 1)
        then goto nullret
        else begin
             rf := (if f[rf + 2] = 0 then rf + 2
                                     else rf + 4 + 2 × sub + uorl);
             setup(rf,ct);
             end;
        goto unstack;
        end  This primitive returns as a signed BCD integer a dimension
             bound value;
```

```
funac:  begin
        integer point;
        real temp;
        if f[rf + 2] = 0
        then point := rf + 5
        else begin
             point := index(rf+3,3);
             if point < 6 + 2 × f[rf +2] or point > f[rf + 3]
             then goto nullret;
             end;
        temp := a[point];
        f[point] := a[point];
        setup(point,ct);
        a[point] := temp;
        goto unstack;
        end  This primitive converts an array element to an integer;
```

```
funae:  begin
        integer put,get;
        getfa(2,0);
        if f[rf + 2] = 0
        then put := 5
        else begin
             put := index(rf + 2,3);
             if put < 6 + 2 × f[rf + 2] or put > f[rf +3]
             then goto nullret;
             end  Put is the location of the element in which we
                  shall store the new value;
        np := rf;
        getfa(3 + f[np + 2],0);
        if f[rf + 2] = 0
        then get := 5
        else begin
             get := index(rf + 2,4 + f[np + 2]);
             if get < 6 + 2 × f[rf + 2] or get > f[rf + 3]
             then goto nullret;
             end  Get is the location of the element to be fetched;
        a[np + put] := a[rf + get];
        goto nullret;
        end  This primitive sets one array element to the value of
             another;



funaf:  begin
        integer i,j,templ;
        getfa(2,5);
        np := rf;
        getfa(3,5);
        nq := rf;
        getfa(4,5);
        templ := f[np + 7] - f[np + 6];
        if f[np + 2] = 2 and f[nq + 2] = 1 and f[rf + 2] = 2
           and templ = f[np + 9] - f[np + 8]
           and templ = f[nq + 7] - f[nq + 6]
           and templ = f[rf + 7] - f[rf + 6]
           and templ = f[rf + 9] - f[rf + 8]
        then begin
             for i := 8 step 1 until f[nq + 3]
             do if a[nq + 7 + i] = 0.
                then zbranch(5);
             for i := 0 step 1 until templ
             do for j := 0 step 1 until templ
                do a[map(rf,i,j)] := a[map(np,i,j)]/
                                     (a[nq + 7 ÷ i] × a[nq + 7 + j]);
             comment  This half does the operation when two matrices and
                      a vector are involved.  Notice the check on the side
                      lengths of the arrays involved and on possibility of
                      a vector element being zero;
             end
```

```
else if f[np + 2] = 0 and f[nq + 2] = 0 and f[rf + 2] = 0
         and a[nq + 5] ≠ 0.
      then a[rf + 5] := a[np + 5]/a[nq + 5]↑2
      else zbranch(5);
      comment  This portion takes care of all the arguments
               being scalars or the standard errors;
      goto nullret;
      end  'funaf' is Beaton routine DMD;


funag:  begin
        integer i,j,templ;
        getfa(2,5);
        nq := rf;
        getfa(3,5);
        np := rf;
        getfa(4,5);
        templ := f[np + 7] - f[np + 6];
        if f[np + 2] = 2 and f[nq + 2] = 1 and f[rf + 2] = 2
           and templ = f[np + 9] - f[np + 8]
           and templ = f[nq + 7] - f[nq + 6]
           and templ = f[rf + 7] - f[rf + 6]
           and templ = f[rf + 9] - f[rf + 8]
        then for i := 0 step 1 until templ
             do for j : 0 step 1 until templ
                do a[map(rf,i,j)] := a[map(np,i,j)] ×
                                     a[nq + 7 + i] × a[nq + 7 + j]
        else if f[np +2] = 0 and f[nq + 2] = 0 and f[rf + 2] = 0
             then a[rf + 5] := a[np + 5] × a[nq + 5]↑2
             else zbranch(5);
        goto nullret;
        end  'funag' is Beaton routine SCP.  It has almost exactly the
             same structure as 'funaf' (DMD), except that the vector
             elements may be zero for this routine;
```

```
funah:  begin
        integer i,j,k,temp;
        real pivot;
        getfa(2,5);
        np := rf;
        getfa(3,5);
        get(4);
        k := convert(rp,rq);
        temp := f[np + 7] - f[np + 6];
        if f[np + 2] = 2 and f[rf + 2] = 2
           and temp = f[np + 9] - f[np + 8]
           and temp = f[rf + 7] - f[rf + 6]
           and temp = f[rf + 9] - f[rf + 9]
           and -1 < k and k < temp + 1
           and a[map(np,k,k)] ≠ 0.
        then begin
             pivot := a[map(np,k,k)];
             for i := 0 step 1 until temp
             do for j := 0 step 1 until temp
                do if i ≠ k and j ≠ k
                   then a[map(rf,i,j)] := a[map(np,i,j)] -
                          (a[map(np,i,k)] - a[map(np,k,j)]/pivot
                   else if i ≠ k and j = k or i = k and j ≠ k
                          then a[map(rf,i,j)] := a[map(np,i,j)]/pivot
                          else a[map(rf,k,k)] := 1/pivot;
             end
        else zbranch(5);
        goto nullret;
        end  'funah' is Beaton routine SWP. The coding is a straight-
             forward translation of the algebra for the routine. Note
             that the routine only operates when the two arguments are
             matrices;


funai:  begin
        integer temp,hold,i,j,k;
        real pivot,mult,temp2,chek;
        integer procedure place(x,y);
          integer x,y;
          value x,y;
              begin
              place : temp × x + y + hold;
              end  There has to be a place to store an additional matrix
                   the size of the one which is to be inverted and also
                   a vector as long as the side of the array. These will
                   be stored in the left hand end of the 'a' vector, and
                   this routine will reference the element subscripted
                   with 'x' and 'y' in this storage area;
```

```
getfa(2,5);
np := rf;
getfa(3,5);
nq := rf;
getfa(4,5);
temp := f[np + 7] - f[np + 6];
if f[np + 2] = 2 and f[nq + 2] = 2 and f[rf + 2] = 0
   and temp = f[np + 9] - f[np + 8]
   and temp = f[nq + 7] - f[nq + 6]
   and temp = f[nq + 9] - f[nq + 8]
   and (temp + 1) × (temp + 2) < fl
then begin
     comment  The conditional checks to see that there are
              actually two square arrays and a scalar and
              that there is enough free storage at the upper
              end of the 'a' vector for the auxiliary vector
              and matrix needed in the inversion;
     hold := fl - (temp + 1) × (temp + 2) + 1;
     for i := 0 step 1 until temp
     do begin
        for j := 0 step 1 until temp
        do begin
           a[map(nq,i,j)] := 0.;
           a[place(i,j)] := a[map(np,i,j)];
           end;
        a[map(nq,i,i)] := 1.;
        end  This little section transfers the original matrix
             to auxiliary storage, so that it will not be harmed
             in the inversion process and places the identity
             matrix in the inverse;
     a[rf + 5] := 1.;
     for i := 0 step 1 until temp
     do begin
        if a[place(i,i)] ≠ 0.
        then goto loop
        else for j := i step 1 until temp
             do if a[place(j,i)] ≠ 0.
                then begin
                     for k := 0 step 1 until temp
                     do begin
                        a[place(i,k)] := a[place(i,k)]
                         + a[place(j,k)];
                        a[map(nq,i,k)] := a[map(nq,i,k)]
                         + a[map(nq,j,k)];
                        goto loop;
                        end  This section replaces a zero pivot
                             element with a non-zero one;
        goto fail;
```

```
loop:              pivot := a[place(i,i)];
                   a[fl - i] := pivot;
                   a[rf + 5] := a[rf + 5] × pivot
                   for j := 0 step 1 until temp
                   do if j ≠ i
                     then begin
                        mult := a[place(j,i)]/pivot;
                        for k := 0 step 1 until temp
                        do begin
                          a[map(nq,j,k)] := a[map(nq,j,k)]
                           - mult × a[map(nq,i,k)];
                          a[place(j,k)] := a[place(j,k)]
                           - mult × a[place(i,k)];
                          end;
                        end;
                   end  This whole begin block does the Gauss elimination
                        for one row each time through the for loop;
                 for i := 0 step 1 until temp
                 do for j : 0 step 1 until temp
                   do a[map(nq,i,j)] := a[map(nq,i,j)]/a(fl - i];
                 comment  This normalizes the inverse by dividing each row
                          by the appropriate pivot element.  This was not
                          done before so that small pivot elements would
                          not affect the array too much before they had to;
test:            for i := 0 step 1 until temp
                 do for j := 0 step 1 until temp
                   do begin
                     a[place(i,j)] := 0.;
                     for k := 0 step 1 until temp
                     do a[place(i,j)] := a[place(i,j)] +
                       a[map(np,i,k)] × a[map(nq,k,j)];
                     end  This forms the product of the original array
                          and the approximated inverse to check for size
                          of the residual.  The check is simply an element
                          by element tolerance check;
                 for i := 0 step 1 until temp
                 do for j := 0 step 1 until temp
                   do begin
                     chek := (if i = j then 1. else 0.);
                     if abs(chek - a[place(i,j)]> .01
                     then goto fixup;
                     end  If this test succeeds on all the elements then
                          the routine is finished.  Otherwise, the inverse
                          is sent to an iterative routine to improve it;
                 goto nullret;
```

```
fixup:   for i := 0 step 1 until temp
         do for j := 0 step 1 until temp
           do begin
             chek := (if i = j then 2. else 0.)
             a[place(i,j)] := chek - a[place(i,j)];
             end  This forms the matrix 2I - A*X;
         for i := 0 step 1 until temp
         do begin
           for j := 0 step 1 until temp
           do a[fl - j] := a[map(nq,i,j)];
           for j : 0 step 1 until temp
           do begin
             a[map(nq,i,j)] := 0.;
             for k := 0 step 1 until temp
             do a[map(nq,i,j)] := a[map(nq,i,j)]
                + a[fl - i] × a[place(k,j)];
             end;
           end  This little section has stored the i-th row of
                inverse in the auxiliary vector and then formed
                a new row of the inverse by multiplying the
                stored row by the matrix we calculated above;
         goto test;
fail:    a[rf + 5] := 0.;
         for i := 0 step 1 until temp
         do for j := 0 step 1 until temp
           do a[map(nq,i,j)] := 0.;
         zbranch(5);
         end  This ends the successful half of the conditional;
     else zbranch(5);
     end  'funai' calculates the inverse and determinant of a given
         matrix by the Gauss elimination method.  If the residual
         after multiplication of the original matrix by the supposed
         inverse is too large, the inverse is improved by the applica-
         tion of the iterative formula
                       Y = X*(2I - A*X),
         where A is the given matrix, X is the attempted inverse and
         Y is an improved estimate.  See Calingaert (1965) for the
         details;
```

```
funaj:  begin
        integer i,temp;
        getfa(2,4);
        np := rf;
        getfa(3,4);
        temp := f[np + 7] - f[np + 6];
        if f[np +2] = 2 and f[rf + 2] = 1
           and temp = f[np + 9] - f[np + 8]
           and temp = f[rf + 7] - f[rf + 6]
        then begin
             for i := 0 step 1 until temp
             do if a[map(np,i,i)] < 0.
                then zbranch(4);
             for i := 0 step 1 until temp
             do a[rf + 7 + i] := sqrt(a[map(np,i,i)]);
             end
        else if f[np + 2] = 0 and f[rf + 2] = 0 and a[np + 5] ≮ 0.
             then a[rf + 5] := sqrt(a[np + 5])
             else zbranch(4);
        goto nullret;
        end  'funaj' is Beaton routine DVEC.  The only interesting
             point is the check so that square roots of negative
             numbers will not be taken;


funak:  begin
        integer i,j,k;
        getfa(2,5);
        np := rf;
        getfa(3,5);
        nq := rf;
        getfa(4,5);
        if f[np + 2] = 1 and f[nq + 2] = 1 and f[rf + 2] = 1
           and f[rf + 7] - f[rf + 6] + 1
              =(f[np + 7] - f[np + 6] + 1)×(f[nq + 7] - f[nq + 6] + 1)
        then begin
             k := 8;
             for i := 8 step 1 until f[np + 3]
             do for j := 8 step 1 until f[nq + 3]
                do begin
                   a[rf + k] := a[np + i] × a[nq + j];
                   k := k + 1;
                   end;
             end
        else zbranch(5);
        goto nullret;
        end  'funak' is Beaton routine DIRPRD.  Notice how the index
             'k' is handled.  It conforms precisely to the algebraic
             definition of the routine;
```

```
funam:  begin
        integer i,j,k,temp1,temp2,temp3,temp4;
        getfa(2,5);
        np := rf;
        getfa(3,5);
        nq := rf;
        getfa(4,5);
        if f[np + 2] = 0 and f[nq + 2] = f[rf + 2]
        then begin
             for i : 1 step 1 until f[nq + 2]
             do if f[nq + 5 + 2 × i] - f[nq + 4 + 2 × i]
                   ≠ f[rf + 5 + 2 × i] - f[rf + 4 + 2 × i]
                then zbranch(5);
             for i := (if f[nq + 2] = 0 then 5 else 6 + 2 × f[nq + 2])
              step 1 until f[nq + 3]
             do a[rf + i] := a[nq + i] + a[np + 5];
             end
        else if f[np + 2] = 1 and f[nq + 2] = 1 and f[rf + 2] = 1
                and f[np + 3] = f[nq + 3]
             then begin
                  a[rf + 5] = 0.;
                  for i := 8 step 1 until f[np + 3]
                  do a[rf + 5] := a[rf + 5] + a[np + i] × a[nq + i];
                  end
             else if f[nq + 2] = 2 and f[nq + 2] = 2 and f[rf + 2] = 2
                     and f[nq + 7] - f[nq + 6] = f[np + 9] - f[np + 8]
                     and f[np + 7] - f[np + 6] = f[rf + 7] - f[rf + 6]
                     and f[nq + 9] - f[nq + 8] = f[rf + 9] - f[rf + 8]
                  then begin
                       temp1 := f[nq + 9] - f[nq + 8]
                       temp2 := f[np + 7] - f[np + 6]
                       temp3 := f[np + 9] - f[np + 8]
                       for i := 0 step 1 until temp1
                       do for j := 0 step 1 until temp2
                          do begin
                             temp3 := map(rf,i,j);
                             a[temp3] := 0.;
                             for k := 0 step 1 until temp4
                             do a[temp3] := a[temp3] +
                                 a[map(np,i,k)]×a[map(nq,k,j)];
                             end;
                       end
                  else zbranch(5);
             goto nullret;
             end  'funam' is the array multiplication routine.  The first
                  section multiplies any array element by element by a
                  scalar, the second section multiplies two vectors and
                  results in a scalar, and the final section multiplies
                  two matrices.  There are checks on the size of subscripts
                  throughout the program.  The coding is pretty much a
                  straight transcription of the algorithm for array multi-
                  plication.
```

```
funas:  begin
        integer dim,length,size,n,i,j,prod;
        getf(2,next);
        goto found;
next:   getfa(2,-1);
found:  if f[rf] = 1
        then goto none;
        delete(rf);
none:   np := rp;
        nq := rq;
        get(3);
        dim := convert(rp,rq);
        if dim + 0
        then length := 6 + rq - rp
        else begin
             if fn - 2 × dim < 1
             then goto diag[1];
             size := 1;
             for n := 1 step 2 until 2 × dim
             do begin
                get(n + 3);
                f[fn - n] := convert(rp,rq);
                get(n + 4);
                f[fn - n + 1] := convert(rp,rq);
                size := size × (f[fn - n] - f[fn - n + 1] + 1);
                end  The amount of storage which needs to be reserved
                     for an array is a function of the dimension bounds
                     for that array and the dimension.  At this point
                     the dimension bounds are calculated and stored
                     immediately to the left of the first form in
                     storage.  If there is not enough room for all of
                     them, there will not be room for the array.  Even-
                     tually, they will be moved into their proper place
                     in the form;
             length := 6 + 2 × dim + size + nq - np;
             end  'length' is the actual length of the form in cells of
                  storage.  The next statement is a test to check if
                  there is room for the array in form storage;
```

```
if fn - length < 1
then goto diag[1];
rf := fn;
fn := fn - length;
f[fn] := 4;
f[fn + 1] := hash(w,np,nq);
f[fn + 3] := dim;
f[fn + 4] := (if dim = 0 then 5 else 5 + 2 × dim + size);
f[fn + 5] := length - 1;
if dim = 0
then a[fn + 6] := 0.
else begin
     f[fn + 6] := 6 + 2 × dim;
     for i := 2 × dim step -1 until 1
     do f[fn + 6 + i] := f[rf + i - 2 × dim];
     comment  This last statement moves the dimension bounds
              from their position at the end of the form into
              the place they will permanently occupy.  In the
              process, it reverses their order, so the sub-
              script bounds on the lowest dimension come first
              in the form.  This organization is required for
              other routines;
     for i := 1 step 1 until dim
     do begin
        prod := 1;
        for j := 1 step 1 until dim - i
        do prod := prod × (f[fn + 6 + 2 × j]
                     - f[fn + 5 + 2 × j] + 1);
        f[fn + 6] := f[fn + 6] - prod ×
                     (f[fn + 7 + 2 × (dim - i)] - 1);
        end  This short segment computes the base value of
             array.  The next several statements set the whole
             array to zeros;
     for i := 6 + 2 × dim + 1 step 1 until f[fn + 4]
     do a[fn + i] := 0.;
     end;
for i := 1 step 1 until nq - np
do f[f[fn + 4 + i]] := w[np + i];
goto nullret;
end  The whole routine is taken somewhat from Mooers' 'insert'
     routine.  It transforms a TRAC array definition into a
     request for storage space.  Notice that there are two kinds
     of arrays, those with zero dimension and those with non-zero
     dimension and that they must be treated separately;
```

```
funat:  begin
        integer i,j;
        getfa(2,4);
        np := rf;
        getfa(3,4);
        if f[np + 2] = 2 and f[rf + 2] = 2
           and f[rf + 9] - f[rf + 8] = f[np + 7] - f[np + 6]
           and f[rf + 7] - f[rf + 6] = f[np + 9] - f[np + 8]
        then for i := 0 step 1 until f[np + 7] - f[np + 6]
             do for j := 0 step 1 until f[np + 9] - f[np + 8]
                do a[map(rf,j,i)] := a[map(np,i,j)];
        else zbranch(4);
        goto nullret;
        end  'funat' is the array transposition routine.  The coding
             is obvious;
```

```
funav:  begin
        getfa(2,0);
        setup(rf + 2,ct);
        goto unstack;
        end  'funav' is the dimension value function.  Note that it
             returns a value;
```

```
funsa:  begin
        getfa(2,4);
        np := rf;
        getfa(3,4);
        if f[np + 2] = 0 and f[rf + 2] = 0
        then a[rf + 5] := abs(a[np + 5]);
        else zbranch(4);
        goto nullret;
        end  'funsa' is the scalar absolute value primitive.  It has
             no complications.
```

```
funse:  begin
        getfa(2,4);
        np := rf;
        getfa(3,4);
        if f[np + 2] = 0 and f[rf + 2] = 0
        then a[rf + 5] := exp(a[np + 5])
        else zbranch(4);
        goto nullret;
        end  'funse' is the scalar exponential function;


funsl:  begin
        getfa(2,4);
        np := rf;
        getfa(3,4);
        if f[np + 2] = 0 and f[rf + 2] = 0 and a[np + 5] > 0.
        then a[rf + 5] := ln(a[np + 5])
        else zbranch(4);
        goto nullret;
        end  'funsl' is the scalar log function;


funsp:  begin
        getfa(2,5);
        np := rf;
        getfa(3,5);
        nq := rf;
        getfa(4,5);
        if f[np + 2] = 0 and f[rf + 2] = 0 and f[nq + 2] = 0
           and a[np + 5] > 0.
        then a[rf + 5] := a[np +5]↑a[nq + 5]
        else if a[np + 5] = 0.
             then a[rf + 5] := 0.
             else zbranch(5);
        goto nullret;
        end  'funsp' is the scalar power function.  At present it
             will only calculate the power of a positive number,
             but this may be changed;
```

```
procedure getfa(n,m);
  integer n,m;
  value n,m;
     begin
     if atb + n > atl
     then zbranch(m);
     get(n);
     find(rp,rq,rf,nullret);
     if f[rf] ≠ 4
     then zbranch(m);
     end  This is the array version of 'getf';



integer procedure index(dimpoint,firstarg);
  integer dimpoint,firstarg;
  value dimpoint,firstarg;
     begin
     integer hold,prod,i,j,base;
     base := dimpoint + 3;
     if f[dimpoint] < 1
     then goto nullret;
     hold := f[base];
     for i := 1 step 1 until f[dimpoint]
     do begin
        prod := 1;
        for j := 1 step 1 until f[dimpoint] - i
        do prod := prod × (f[base + 2 × j] - f[base + 2 × j - 1] + 1);
        get(firstarg + f[dimpoint] - i);
        hold := hold + prod × (convert(rp,rq) - 1);
        end;
     index := hold;
     end  This routine calculates the linear subscript of an array
          element when the array subscripts are in BCD in the neutral
          string.  'dimpoint' is a pointer to the dimension of the
          array which is referenced and 'firstarg' is the number of
          the first array subscript in the argument stack for the
          present function being evaluated;
```

```
integer procedure map(point,i,j);
  integer point,i,j;
  value point,i,j;
     begin
     map := (f[point + 7] - f[point + 6] × (f[point + 8] + i - 1)
             + f[point + 6] + j - 1 + f[point + 5] + point;
     end  This function calculates the linear subscript of a matrix
          element when supplied with a pointer to the head of the
          matrix and two subscripts.  The subscripts must be normalized
          to lie between zero and the range of the dimension which they
          subscript;
```

```
procedure setup(pointer,aorn);
  integer pointer,aorn;
  value pointer,aorn;
     begin
     integer start,i;
     if aorn = 0
     then for i := expand(pointer) step 1 until wl
          do begin
             w[wn] := w[i];
             wn := wn + 1;
             end
     else if aorn = 1
          then begin
               start := expand(pointer);
               if fl - wl - 1 + start < 1
               then goto diag[3];
               comment  This means that the active string is not long
                        enough to handle the string;
               for i := wl step -1 until start
               do begin
                  fl := fl - 1;
                  f[fl] := w[i];
                  end;
               end
          else zbranch(0);
     end  'setup' returns integers from machine language format to BCD
          in either the neutral or active string, depending on the value
          of 'aorn'.  'pointer' refers to the 'f' storage vector;
```

```
procedure zbranch(n);
  integer n;
  value n;
     begin
     if n > 0
     then begin
          get(n);
          goto argret;
          end  If this branch is taken, then the n-th argument in the
               neutral stack is returned as a value;
     else if n = -1
          then goto none
          else goto nullret;
     end  The last two branches provide, first, a special branch for
          the 'delete' routine, and second, a branch to 'nullret'.
          This routine is intended for use in situations where the
          number describing the type of branch may need to be calcu-
          lated;
```

APPLICATIONS

The easiest application is a routine to transform curvilinear models into linear ones. If observations of n predictor variables and m predicted variables are taken, then a curvilinear model for the dependence is function f of the form

$$f(\langle x_1,x_2,\ldots,x_n\rangle) = \langle y_1,y_2,\ldots,y_m\rangle$$

where the vector $\underline{y}$ is not a linear function of the vector $\underline{x}$. It is possible to write, by inspection of the function f, a set of functions g_i, where i ranges from 1 to p, and where p is greater than or equal to n, such that a new set of predictor variables x_i' are generated by

$$g_i(\langle x_1,x_2,\ldots,x_n\rangle) = x_i'$$

and a function f′ such that

$$f'(\langle x_1',x_2',\ldots,x_p'\rangle) = \langle y_1,y_2,\ldots,y_m\rangle = f(\langle x_1,x_2,\ldots,x_n\rangle)$$

and f′ is linear. The functions g_i are not canonically determined by the structure of f, but anybody using a curvilinear model will know enough to determine them in a proper manner and they and the function f′ always exist. It is relatively simple to design a TRAC procedure which, given the transformation functions g_i in some standard notation and the name of a data matrix, will parse the functions and produce a transformed data matrix of the proper dimensions. The scalar absolute value, exponential, logarithm, and power functions were included in the package to make these transformations possible.

Another obvious application is the construction of a statistical desk calculator. This calculator will include built-up TRAC procedures for the standard statistics and the opportunity to use the primitives in TRAC for extraordinary problems. There will be elaborate error recovery and detection routines and materials designed for the guidance of beginning users. This calculator will be relatively slow in comparison to hand-tailored routines for each individual statistic, but this will be made up by the flexibility of the package and the chance for the user to investigate the statistical behavior of the data he is using. Beaton has suggested that this kind of ability is the next step statistics must take in its growth as a research tool.

CONCLUSION

This paper has shown that the addition of statistical powers to TRAC is feasible, and, once the design problem is solved, fairly easy. The problem of combining these operators in usable ways is left to the statistician, although that is fairly easy to do also, once Beaton (1964) is read and understood. But there is more to the problem than this summary suggests.

In the first place, the operators which have to do only with the manipulation of arrays have been arranged so that they may be extended, and in some cases already are, to arrays of higher dimension than two. The array storage mechanisms and the input and output routines are also applicable to the arrays of higher dimension. It is my hope that these all may be combined into a matrix and tensor calculator, which, using the textual nature of TRAC, will be able to manipulate these objects symbolically, and which, using my primitives and extensions of them, will be able to translate these symbolic manipulations in actual arithmetic calculation. TRAC is ideal for interpretive work with textual objects and this is precisely the kind of work that needs to be done in the newly opened field of formal manipulation by machine of algebraic structures. Although the work will not go on at ISVD, I would hope that others could take it up.

A more important point is the transient nature of this solution to ISVD problems in statistics. Earlier, it was mentioned that no primitive in TRAC relies on any other. Thus, although these primitives work for ISVD, there is no reason for them to work for anybody else and there is no reason for anyone else to use them. But there is a reason for their presentation. As others need statistical powers in TRAC, they may add the whole package, a part of it, or only take the suggestions here as a starting point for their own solution. And this is the power of TRAC. Each interpreter may be tailored to the needs of the situation in which it is to be used. This has been on the surface a study of a particular problem for ISVD, but more deeply, it is an essay on the flexibility of TRAC. It is my hope that it will serve more as an example in the philosophy of programming languages and the practical application of that philosophy than as a strict answer to a set problem.

BIBLIOGRAPHY

Arden, B.; Galler; and Graham. (1966) *The Michigan Algorithm Decoder (The MAD Manual).* Revised by E. Organick. Ann Arbor, Michigan: University of Michigan Press.

Beaton, A. (1964) *The Use of Special Matrix Operators in Statistical Calculus.* Research Bulletin RB-64-51. Princeton, New Jersey: Educational Testing Service.

__________. (1966) "Considerations in the Construction of a Computer Language for Data Analysis." Princeton, New Jersey: Educational Testing Service. [unpublished]

Hutchinson, T. (1967) Internal Memoranda on the Beaton Subroutines. Cambridge, Massachusetts: Information System for Vocational Decisions. [unpublished]

LaBrie, R. (1966) "The Feasibility of Providing Statistical Power to a Text Handling Language Used in a Computer Based Educational System." Submitted to Harvard Graduate School of Education, Course B-60.

Mooers, C. (1966) "TRAC, A Procedure-Describing Language for the Reactive Typewriter." *Communications of the ACM.* 9,3.

__________; and Deutsch, P. (1965) "TRAC, A Procedure Handling Language." *Proceedings of the ACM - 20th Annual Conference.* Cleveland, Ohio: 229-246.

__________. (1966) "TRAC in ALGOL, Level Zero Standard Processor (Draft)." Cambridge, Massachusetts: Rockford Research Institute, Inc. [unpublished]

Tiedeman, David V. (1965) "A Proposal for an Information System for Vocational Decisions." Submitted to the U.S. Office of Education by the Harvard Graduate School of Education.

Calingaert, C. (1965) *Principles of Computation.* Reading, Massachusetts: Addison-Wesley.

APPENDIX G

Flow Charts

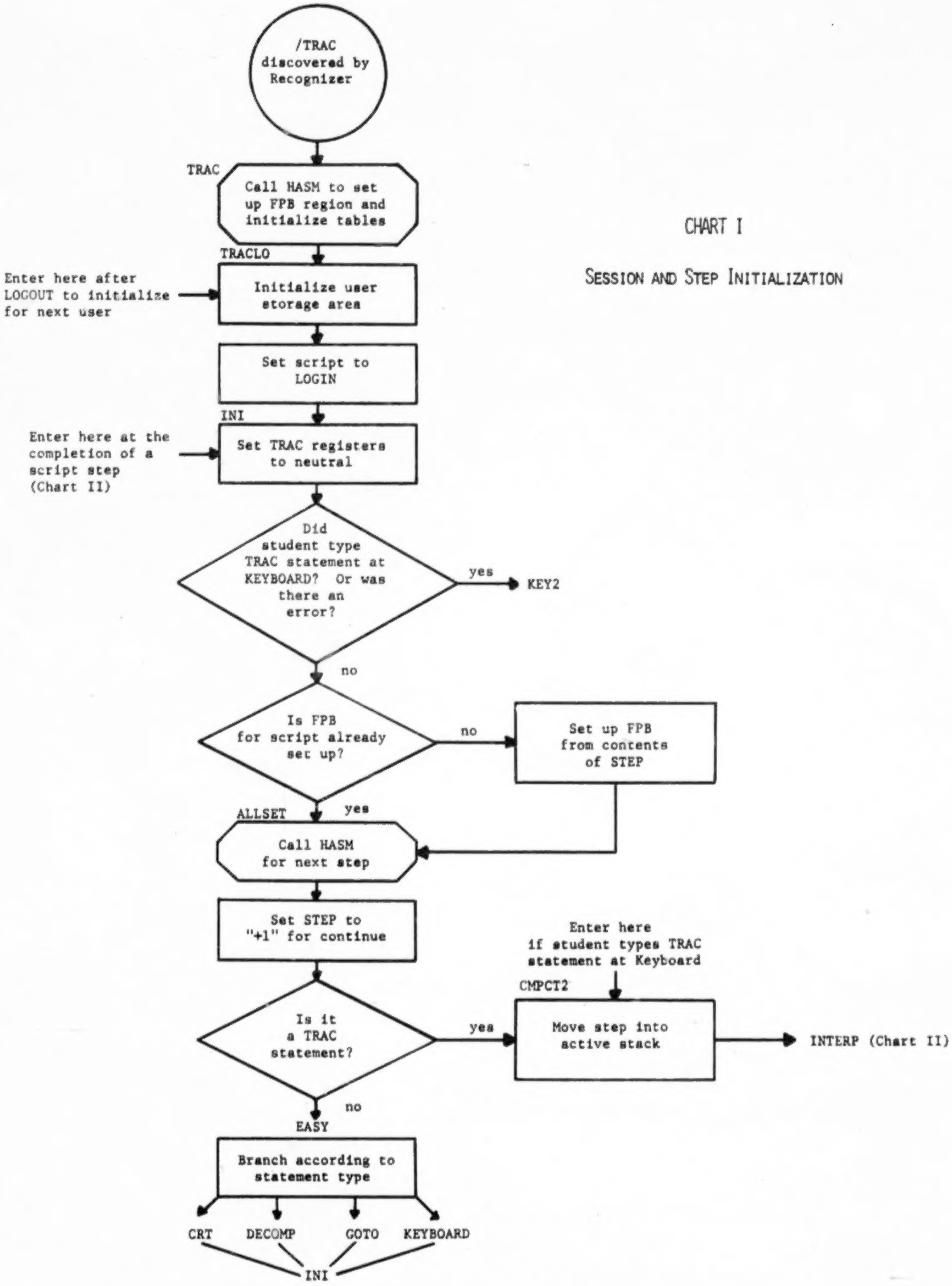

/TRAC discovered by Recognizer
TRAC
Call HASM to set up FPB region and initialize tables
TRACLO
Enter here after LOGOUT to initialize for next user
Initialize user storage area
Set script to LOGIN
INI
Enter here at the completion of a script step (Chart II)
Set TRAC registers to neutral
Did student type TRAC statement at KEYBOARD? Or was there an error?
yes
KEY2
no
Is FPB for script already set up?
no
Set up FPB from contents of STEP
yes
ALLSET
Call HASM for next step
Set STEP to "+1" for continue
Enter here if student types TRAC statement at Keyboard
CMPCT2
Is it a TRAC statement?
yes
Move step into active stack
INTERP (Chart II)
no
EASY
Branch according to statement type
CRT
DECOMP
GOTO
KEYBOARD
INI

CHART I

SESSION AND STEP INITIALIZATION

CHART II

TRAC MAIN INTERPRETER

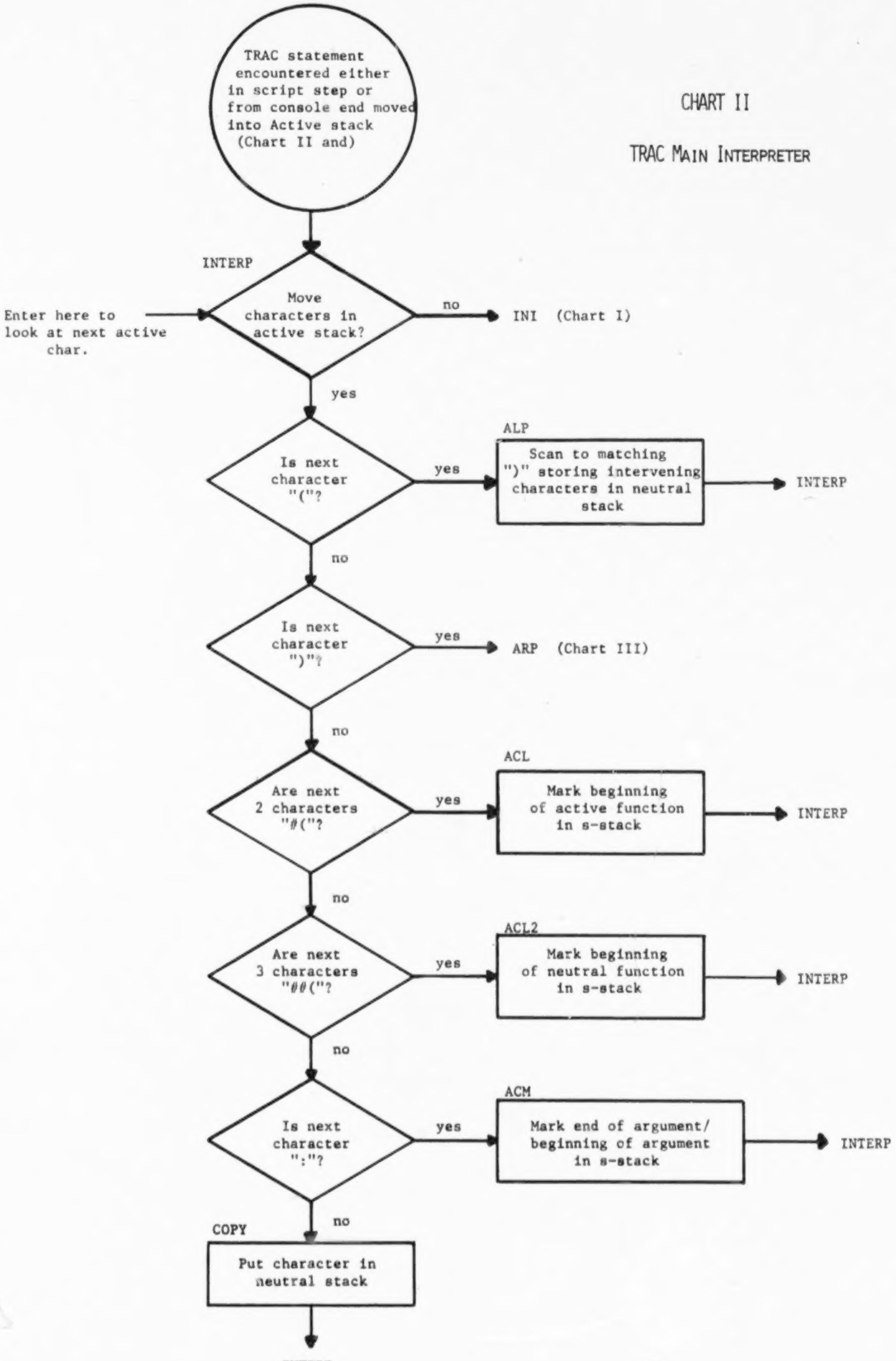
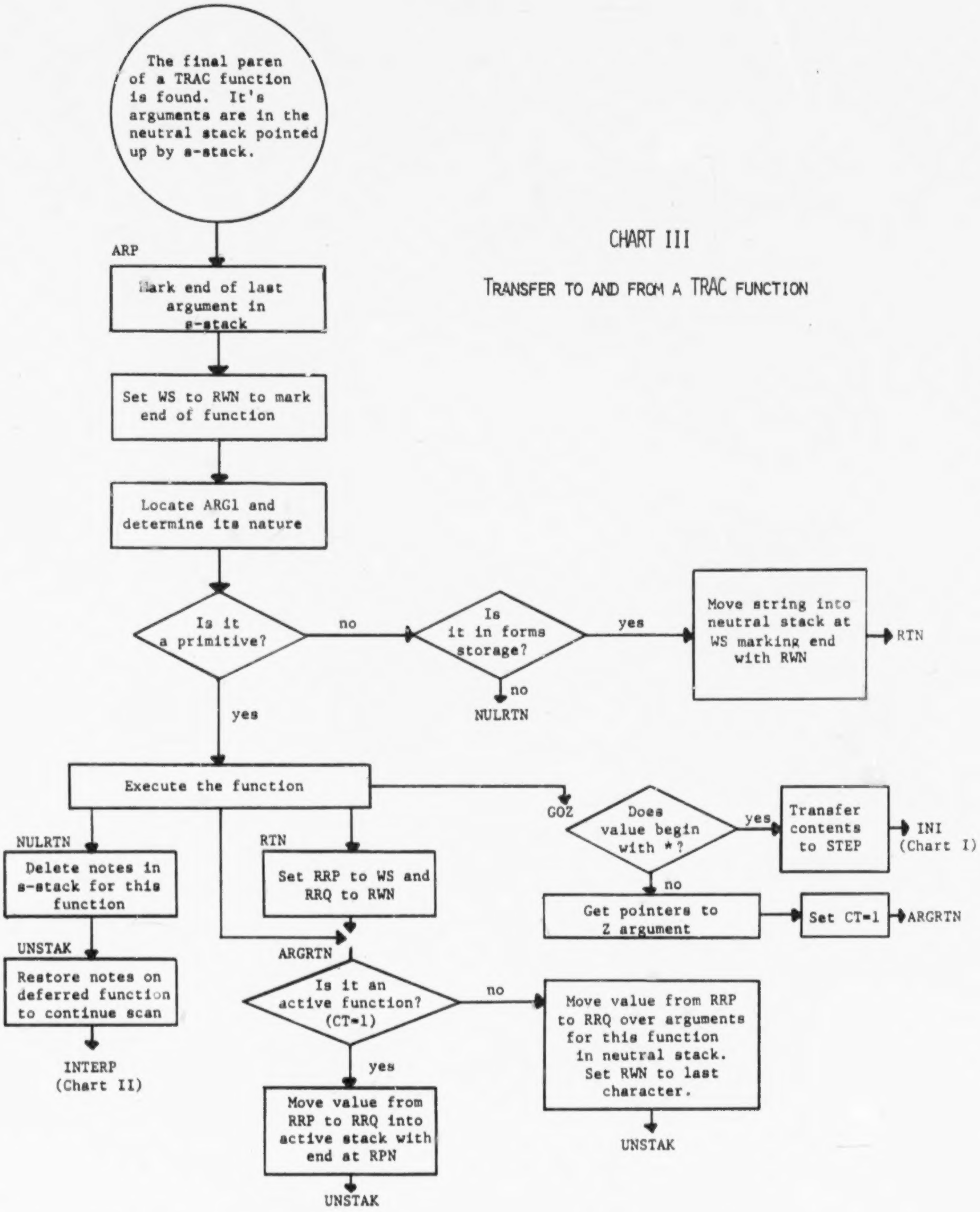

TRAC statement encountered either in script step or from console end moved into Active stack (Chart II and)
INTERP
Enter here to look at next active char.
Move characters in active stack?
no
INI (Chart I)
yes
ALP
Is next character "("?
yes
Scan to matching ")" storing intervening characters in neutral stack
INTERP
no
Is next character ")"?
yes
ARP (Chart III)
no
ACL
Are next 2 characters "#("?
yes
Mark beginning of active function in s-stack
INTERP
no
ACL2
Are next 3 characters "##("?
yes
Mark beginning of neutral function in s-stack
INTERP
no
ACM
Is next character ":"?
yes
Mark end of argument/ beginning of argument in s-stack
INTERP
no
COPY
Put character in neutral stack
INTERP

CHART III

TRANSFER TO AND FROM A TRAC FUNCTION


The final paren of a TRAC function is found. It's arguments are in the neutral stack pointed up by s-stack.
ARP
Mark end of last argument in s-stack
Set WS to RWN to mark end of function
Locate ARG1 and determine its nature
Is it a primitive?
no
Is it in forms storage?
yes
Move string into neutral stack at WS marking end with RWN
RTN
no
NULRTN
yes
Execute the function
GOZ
Does value begin with *?
yes
Transfer contents to STEP
INI (Chart I)
no
Get pointers to Z argument
Set CT=1
ARGRTN
NULRTN
Delete notes in s-stack for this function
UNSTAK
Restore notes on deferred function to continue scan
INTERP (Chart II)
RTN
Set RRP to WS and RRQ to RWN
ARGRTN
Is it an active function? (CT=1)
no
Move value from RRP to RRQ over arguments for this function in neutral stack. Set RWN to last character.
UNSTAK
yes
Move value from RRP to RRQ into active stack with end at RPN
UNSTAK